Week 7

DP: LAS, Subset Sum, Knapsack

Thorben Klabunde

www.th-kl.ch

November 3, 2025

Agenda

1. Mini-Quiz
2. Assignment
3. Recap
4. 1) Sequence Alignment
5. 2) Sequential Decision
6. 3) Bounded Resources
7. Summary
8. Additional Practice
9. Peer Grading

Mini-Quiz

Q1

In a linked list data structure L, suppose you have a pointer to an object o in L. Then inserting a key k into the list after o (i.e. $\texttt{insertAfter}(o, k, L)$) takes $O(1)$ time.

- (•) True ✓
- ( ) False

Since we have a pointer to the object o, we can replace o's subsequent key with k and then reattach the rest of the list to be after k. Since we are just moving around a couple pointer values, this all takes $O(1)$ time.

The correct answer is 'True'.

Q2

Let T be a 2-3 tree with depth 4. Let x be the number of leaves. It is possible for x to be equal to:

True	False			
○ ⊗	◉ ✓	5	✓	
○ ⊗	◉ ✓	10	✓	
◉ ✓	○ ⊗	20	✓	
◉ ✓	○ ⊗	50	✓	
○ ⊗	◉ ✓	100	✓	
○ ⊗	◉ ✓	200	✓	
○ ⊗	◉ ✓	1000	✓	

Q2

If every node has 2 children, then a 2-3 tree of depth 4 has $2^4 = 16$, this is the minimum number of leaves. If every node has 3 children then depth 4 means we have $3^4 = 81$ leaves, this is the maximum number of leaves. Its then possible to achieve every number in this range by having a mix of degree 2 and degree 3 nodes.

5: False
10: False
20: True
50: True
100: False
200: False
1000: False

Q3

Let $B1, B2$ be two leaves in a 2-3-tree. If an insert operation increases the depth of $B1$, then it also increases the depth of $B2$.

- (•) True ✓
- ( ) False

All leaves must be the same depth, so increasing the depth of one leaf must increase the others too.

The correct answer is 'True'.

Q4

We can find the maximum key of a 2-3 tree with n leaves in time $O(\log(n))$.

- (•) True ✓
- ( ) False

We can keep going down the right subtree to get the maximum key, which will be of depth $\log(n)$.

The correct answer is 'True'.

Q5

Consider the subset sum problem with input $A[1 \ldots n]$ and target value b, where the subproblem $T(i, s)$ denotes whether s is a subset sum of $A[1 \ldots i]$. Fill in X such that the recursion is $T(i, s) = T(i-1, s) \vee T(i-1, X)$ (for $2 \leq i \leq n$). If $X < 0$ then $T(i-1, X)$ is considered to be false.

a. $X = A[i-1]$

b. $X = s - A[i]$ ✓

c. $X = s - b$

d. $X = b$

Q5

Your answer is correct.

Achieving a subset sum of s can be done in two ways corresponding to the two conditions. Either we can make it by not including the element at index i, which represents $T(i-1, s)$. OR we include the element at index i in which case we need to check that we can make a subset sum of $s - A[i]$ with elements up to index $i-1$, corresponding to $T(i-1, s-A[i])$.

The correct answer is: $X = s - A[i]$

Q6

There is a known algorithm which solves subset sum in time $O(n^3)$.

- True
- False ✓

It was discussed in lecture that if there was a polynomial time algorithm for subset sum, then $P = NP$. This is a major unknown open problem.

The correct answer is 'False'.

Q7

Consider the knapsack problem with weights w_i, profits p_i and weight limit W. Let $P = p_1 + \ldots + p_n$. Which of the following statements are true/false:

True	False			
		There is an algorithm solving the problem in time $O(nW)$.		
		There is an algorithm solving the problem in time $O(nP)$.		

Q7

Both are true. For $O(nP)$ we saw in lecture a 2 dimensional DP with table entries $G[i, p] =$ the minimum weight needed to achieve profit at least p with items $1 \dots i$.
For $O(nW)$ we saw in lecture a 2 dimensional DP with table entries $P[i, w] =$ maximum profit achievable with weight w with items $1 \dots i$.

There is an algorithm solving the problem in time $O(nW)$.
: True
There is an algorithm solving the problem in time $O(nP)$.
: True

Q8

There is an algorithm for knapsack which computes in polynomial time a solution which is at least half as good as the optimal solution.

- (•) True ✓
- ( ) False

In lecture we saw a polynomial algorithm that computes a $(1-\epsilon)$-approximation to the knapsack problem in time $O(n^3\epsilon^{-1})$. Taking $\epsilon = 1/2$, we get a $1/2$-approximation which runs in time $O(2n^3)$ which is polynomial in n.

The correct answer is 'True'.

Q9

For the knapsack problem, let OPT be the optimal solution for the original profits p_i and $\widetilde{OPT}$ the optimal solution for the rounded profits $\tilde{p}_i$. Which of the following is always true?

a. $\sum_{i \in \widetilde{OPT}} \tilde{p}_i \geq \sum_{i \in OPT} \tilde{p}_i$. ✓

b. $\sum_{i \in \widetilde{OPT}} \tilde{p}_i = \sum_{i \in OPT} \tilde{p}_i$.

c. $\sum_{i \in \widetilde{OPT}} \tilde{p}_i \leq \sum_{i \in OPT} \tilde{p}_i$.

d. None of the above.

Q9

Your answer is correct.

Note that $\sum_{i \in \widetilde{OPT}} \tilde{p}_i$ is the optimal (maximal profit) choice of elements for the profits $\tilde{p}_i$ which satisfy the weight limit. So as OPT also satisfies the weight limit, we know that the profit from $\sum_{i \in OPT} \tilde{p}_i$ cannot be greater.

The correct answer is: $\sum_{i \in \widetilde{OPT}} \tilde{p}_i \geq \sum_{i \in OPT} \tilde{p}_i$.

Q10

Let L be a longest ascending subsequence of $A[1 \dots i]$ and L' be a longest ascending subsequence of $A[1 \dots i+1]$. Then L' is either identical to L, or L' is obtained from L by adding $A[i+1]$ at the end.

- True
- False ✓

Consider $A = [1, 3, 2]$, then for $A[1 \dots 2]$ there is only 1 longest ascending subsequence $[1, 3]$. But for $A[1 \dots 3]$ there is both $[1, 3]$ and $[1, 2]$ and note that the second is not $[1, 3]$ or some addition to that subsequence.

The correct answer is 'False'.

Assignment

Feedback Assignment 5

Common points from last week's assignment:

- **Function Calls:** Well done! Generally just small remarks: Be careful with notation (redefining already defined terms, writing ill-defined terms)
- **Bubble Sort Proof:** Like mentioned last week, make sure that you explain why/how the algorithm achieves the correct state. Ensure that your invariant captures all required information to allow the final conclusion. E.g., the given algorithm only had n-1 iterations, so we need to add the additional information that the last j elements are the largest elements to ensure correct ordering of the first element at the end.

Remember to check the detailed feedback on Moodle! Reach out if you have any questions regarding the corrections.

Assignment 6

- We will have a **closer look at Ex. 6.3 and Ex. 6.4** today.
- The remaining exercises are **covered in detail** in the **master solution**. If you have any questions, **don't hesitate to reach out**!

Ex. 6.3

Exercise 6.3 *Introduction to dynamic programming* **(1 point)**.

Consider the recurrence

$$A_1 = 1$$
$$A_2 = 2$$
$$A_3 = 3$$
$$A_4 = 4$$
$$A_n = A_{n-1} + A_{n-3} + 2A_{n-4} \text{ for } n \geq 5.$$

(a) Provide a recursive function (using pseudo code) that computes A_n for $n \in \mathbb{N}$. You do not have to argue correctness.

Solution:

Algorithm 1 $A(n)$

if $n \leq 4$ **then**
  return n
else
  return $A(n-1) + A(n-3) + 2A(n-4)$

Ex. 6.3b)

(b) Lower bound the run time of your recursion from (a) by $\Omega(C^n)$ for some constant $C > 1$.

Informal sketch:

Let $T(n)$ be the number of operations of a call to $A(n)$ with $n \in \mathbb{N}$ and assume $n \geq 4$.

Notice that that each recursive call performs a constant number of operations and we get closer to the base cases with each call. We can thus assume that $T(n)$ is monotonically increasing.

Now W.l.o.g. assume that n is divisible by 4 (take the next smaller multiple of 4 otherwise).

Then: $T(n) = T(n-1) + T(n-3) + 2T(n-4) \geq 4 \cdot T(n-4) \geq \ldots \geq 4^{(\frac{n}{4}-1)} \cdot \underbrace{T(4)}_{1} \geq \Omega(C^n)$ for $C = 4^{1/4}$

Make this argument formal by proving $T(n) \geq \frac{1}{4} 4^{n/4}$ by induction (see master solution).

Ex. 6.3c)

(c) Improve the run time of your algorithm using memoization. Provide pseudo code of the improved algorithm and analyze its run time.

Ex. 6.3c)

Algorithm 2 Compute A_n using memoization

```
memory ← n-dimensional array filled with (−1)s
function A_Mem(n)
    if memory[n] ≠ −1 then                          ▷ If A_n is already computed.
        return memory[n]
    if n ≤ 4 then
        memory[n] ← n
        return n
    else
        A_n ← A_Mem(n − 1) + A_Mem(n − 3) + 2A_Mem(n − 4)
        memory[n] ← A_n
        return A_n
```

Ex. 6.4

Exercise 6.4 *Coin Conversion* **(1 point).**

Suppose you live in a country where the transactions between people are carried out by exchanging coins denominated in dollars. The country uses coins with k different values, where the smallest coin has value of $b_1 = 1$ dollar, while other coins have values of $b_2, b_3, \ldots, b_k$ dollars. You received a bill for n dollars and want to pay it *exactly* using the smallest number of coins. Assuming you have an unlimited supply of each type of coin, define OPT to be the minimum number of coins you need to pay exactly n dollars. Your task is to calculate OPT. All values $n, k, b_1, \ldots, b_k$ are positive integers.

Example: $n = 17, k = 3$ and $b = [1, 9, 6]$, then OPT $= 4$ because 17 can be obtained via 4 coins as $1 + 1 + 9 + 6$. No way to obtain 17 with three or less coins exists.

(a) Consider the pseudocode of the following algorithm that "tries" to compute OPT.

Algorithm 3

```
Input: integers n, k and an array b = [1 = b_1, b_2, b_3, ..., b_k].

counter ← 0
while n > 0 do
    Let b[i] be the value of the largest coin b[i] such that b[i] ≤ n.
    n ← n − b[i].
    counter ← counter + 1
Print("min. number of required coins = ", counter)
```

greedy approach!

Algorithm 3 does not always produce the correct output. Show an example where the above algorithms fails, i.e., when the output does not match OPT. Specify what are the values of n, k, b, what is OPT and what does Algorithm 3 report.

Note, the greedy approach does not guarantee an optimal solution here.

Counterexample:

Let $n=12$, $k=3$, $b=[1,8,6]$.

Alg. 3 returns 4 $(8,1,1,1)$

but the optimal solution is 2 $(6,6)$.

Ex. 6.4b)

(b) Consider the pseudocode below. Provide an upper bound in O notation that bounds the time it takes a compute $f[n]$ (it should be given in terms of n and k). Give a short high-level explanation of your answer. For full points your upper bound should be tight (but you do not have to prove its tightness).

Algorithm 4

```
Input: integers n, k. Array b = [1 = b_1, b_2, b_3, ..., b_k].

Let f[1...n] be an array of integers.
f[0] ← 0                                          ▷ Terminating condition.
for N ← 1...n do
    f[N] ← ∞                                      ▷ At first, we need ∞ coins. We try to improve upon that.
    for i ← 1...k do
        if b[i] ≤ N then
            val ← 1 + f[N − b[i]]                 ▷ Use coin b[i], it remains to optimally pay N − b[i].
            f[N] ← min(f[N], val)
Print(f[n])
```

Ex. 6.4d)

(d) Rewrite Algorithm 4 to be recursive and use memoization. The running time and correctness should not be affected.

Notice that Alg. 4 solves the subproblem: $f[i]$ = min. number of coins to make sum of i.

Algorithm 4

```
1: Input: integers n, k. Array b = [1 = b_1, b_2, b_3, ..., b_k].
2:
3: Let f[1...n] be an array of integers.
4: f[0] ← 0                                   ▷ Terminating condition.
5: for N ← 1...n do
6:     f[N] ← ∞                               ▷ At first, we need ∞ coins. We try to improve upon that.
7:     for i ← 1...k do
8:         if b[i] ≤ N then
9:             val ← 1 + f[N − b[i]]          ▷ Use coin b[i], it remains to optimally pay N − b[i].
10:            f[N] ← min(f[N], val)
11: Print(f[n])
```

Runtime: There are n different states to be computed, each in $O(k)$ due to the inner for-loop.

$\Rightarrow O(n \cdot k)$ in total.

Ex. 6.4c)

(c) Let $\mathrm{OPT}(N)$ be the answer (min. number of coins needed) when $n = N$. Algorithm 4 (correctly) computes a function $f[N]$ that is equal to $\mathrm{OPT}(N)$. Formally prove why this is the case, i.e., why $f[N] = OPT(N)$.

Hint: *Use induction to prove the invariant $f[n] = OPT(n)$. Assume the claim holds for all values of $n \in \{1, 2, \ldots, N-1\}$. Then show the same holds for $n = N$.*

Ex. 6.4c)

(c) Let $\mathrm{OPT}(N)$ be the answer (min. number of coins needed) when $n = N$. Algorithm 4 (correctly) computes a function $f[N]$ that is equal to $\mathrm{OPT}(N)$. Formally prove why this is the case, i.e., why $f[N] = OPT(N)$.

Hint: *Use induction to prove the invariant $f[n] = OPT(n)$. Assume the claim holds for all values of $n \in \{1, 2, \ldots, N-1\}$. Then show the same holds for $n = N$.*

Sketch:

We proceed by induction on n.

B.C.: Let $n = 0$. Notice that $f[0] = 0 = OPT(0)$ and the B.C. trivially holds.

I.H.: Assume for some $N \in \mathbb{N}$ and all $n \in \{1, \ldots, N-1\}$ that $f[n] = OPT(n)$.

Be careful, here we require a strong induction!

Ex. 6.4c)

(c) Let $\mathrm{OPT}(N)$ be the answer (min. number of coins needed) when $n = N$. Algorithm 4 (correctly) computes a function $f[N]$ that is equal to $\mathrm{OPT}(N)$. Formally prove why this is the case, i.e., why $f[N] = OPT(N)$.

Hint: *Use induction to prove the invariant $f[n] = OPT(n)$. Assume the claim holds for all values of $n \in \{1, 2, \ldots, N-1\}$. Then show the same holds for $n = N$.*

I.S.: It remains to prove that $f[N] = OPT(N)$.
Notice that the inner for-loop maintains a minimum over the set $\{1 + f[N-\delta_i] \mid i \le k \text{ s.t. } N-\delta_i \ge 0\}$
$\Rightarrow 1 + f[N-\delta_i] \ge f[N]$, for all $i \in [k]$.

Suppose for sake of contradiction that ...

(1) $f[N] > OPT(N)$:
$\Rightarrow 1 + f[N-\delta_i] > OPT(N)$, for all $i \in [k]$.
But clearly $OPT(N) = OPT(N-\delta_i) + 1$, for some $i \in [k]$. Hence, $f[N-\delta_i] > OPT(N-\delta_i)$ for some $i \in [k]$, a contradiction to our I.H. Hence $f[N] \le OPT(N)$.

(2) $f[N] < OPT(N)$:
$\Rightarrow 1 + f[N-\delta_i] < OPT(N)$, for some $i \in [k]$
$\Rightarrow OPT(N-\delta_i) \overset{IH}{=} f[N-\delta_i] < OPT(N) - 1$, for some $i \in [k]$ $\Rightarrow OPT(N)$ is not optimal, a contradiction.

It follows that $f[N] = OPT(N)$. ∎

Ex. 6.4d)

Algorithm 5

```
1:  Input: integers n, k. Array b = [1 = b_1, b_2, b_3, ..., b_k].
2:  Global variable: memo[1...n], initialized to -1.
3:
4:  function f(N)
5:      if N = 0 then return 0
6:      if memo[N] ≠ -1 then return memo[N]
7:      solution ← ∞
8:      for i ← 1...k do
9:          if b[i] ≤ N then
10:             val ← 1 + f(N - b[i])          ▷ Use coin b_i, it remains to optimally pay x - b_i.
11:             solution ← min(solution, val)   ▷ Check if this is the best seen so far?
12:     memo[N] ← solution
13:     return solution
14:
15: Print("OPT = ", f(n))
```

Ex. 6.4d)

Again, we write our code under the assumption that we have the solution for $N - b_i$, for all $i \in [k]$, which is returned by the call to $f(N - b_i)$.

Algorithm 5

```
1:  Input: integers n, k. Array b = [1 = b_1, b_2, b_3, ..., b_k].
2:  Global variable: memo[1...n], initialized to −1.
3:
4:  function f(N)
5:      if N = 0 then return 0
6:      if memo[N] ≠ −1 then return memo[N]
7:      solution ← ∞
8:      for i ← 1...k do
9:          if b[i] ≤ N then
10:             val ← 1 + f(N − b[i])          ▷ Use coin b_i, it remains to optimally pay x − b_i.
11:             solution ← min(solution, val)   ▷ Check if this is the best seen so far?
12:     memo[N] ← solution
13:     return solution
14:
15: Print("OPT = ", f(n))
```

Recap

Recall - The DP "Recipe":

A general plan for solving DP problems:

1. **Define a suitable subproblem.**
   - What is the structure of the solution? What are the parameters? (e.g., $DP[i]$, $DP[i][j]$)
2. **Establish a recurrence relation.**
   - How does the solution to a problem depend on smaller subproblems?
   - Define base cases!
3. **Compute the solutions (Bottom-Up).**
   - In what order must the DP table be filled?
4. **Assemble the solutions.**
   - Where in the table is the final solution?
5. **Analyze correctness and runtime.**

Why Look for Patterns?

As I mentioned last week, DP is about analyzing structure.

- The "trick" to DP is **finding the right subproblem**.
- Different problem structures lead to different kinds of subproblems.
- We aren't memorizing solutions, we are learning **patterns of thought**.
- By comparing the problems, we can identify "archetypes" that help narrow down the search space.

Key Insight

The structure of your subproblem should match the **decision structure** of your problem.

- **What choices do you make** at each step?
- **What information do you need** to make those choices?

Our DP Problem "Archetypes"

Most DP problems we've seen fall into one of three main families:

Sequence Alignment / "Prefix Comparison" - *Last Week*

- **Core Idea:** Compare two sequences, A and B.
- **Subproblem:** $DP[i][j] =$ Solution for prefixes $A[1..i]$ and $B[1..j]$.
- **Examples:** Longest Common Subsequence, Edit Distance.

Sequential Decision / "Ending At" - *Last Week*

- **Core Idea:** Build an optimal structure in a single sequence where order matters.
- **Subproblem:** $DP[i] =$ Solution *ending at* index i (or reaching i).
- **Examples:** LAS, Jump Game, Max Subarray.

Bounded Resource / "Take It or Leave It"

- **Core Idea:** Select an optimal subset of items with a constraint.
- **Subproblem:** $DP[i][k] =$ Solution using items $1..i$ with resource k.
- **Examples:** $0/1$ Knapsack, Subset Sum.

Connecting to This Week's Problems

Let's map this week's problems to the archetypes:

- **Longest Ascending Subsequence:** Archetype 2 (Sequential Decision)
  - Single array $A[1..n]$ with structural rule: ascending order
  - Decision: "Which valid predecessor $j < i$ (where $A[j] < A[i]$) extends best?"
- **Subset Sum:** Archetype 3 (Bounded Resource)
  - Items $A[1..n]$ + target sum constraint b
  - Binary choice: Take $A[i]$ or leave it.
- **0/1 Knapsack:** Archetype 3 (Bounded Resource)
  - Items with weights/profits + capacity constraint W
  - Binary choice: Take item i (and pay w_i) or leave it.

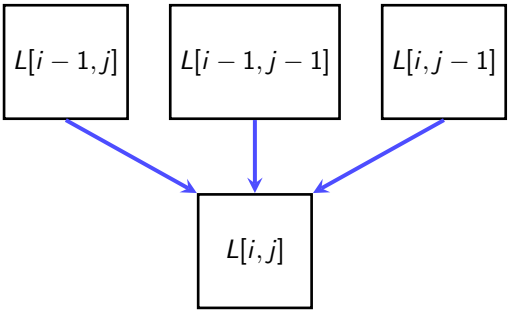
1) Sequence Alignment

Archetype 1: Sequence Alignment Problems (Last Week)

- **Archetype:** Longest Common Subsequence
- **Key Feature: Alignment**. Find the best way to match, skip, or transform elements from two sequences.
- **Subproblem:** $L[i][j]$ = LCS of $A[1..i]$ and $B[1..j]$.
- **Key Idea:** The solution for $L[i][j]$ depends only on decisions about the last characters, $A[i]$ and $B[j]$.
- **Recurrence:**

$$L[i,j] = \begin{cases} 1 + L[i-1, j-1] & \text{if } a_i = b_j \\ \max(L[i-1,j], L[i,j-1]) & \text{if } a_i \neq b_j \end{cases}$$

Depends only on its 3 neighbors



Archetype 1: Why 2D?

Intuition for the Subproblem:

- We cannot solve the problem without knowing our position in *both* sequences.
- This forces us to use a 2D state: $DP[i][j] =$ "Solution for $A[1..i]$ and $B[1..j]$".

Why 1D Fails for LCS

Try: $L[i] =$ LCS length for $A[1..i]$ and $B[1..i]$

Problem: The optimal LCS might use different positions from each string:

E.g., $A =$ "TIGER", $B =$ "ZIEGE"

Optimal LCS "IGE" uses $A[2, 3, 5]$ and $B[2, 3, 4]$

We **lose crucial information** about where we are in each sequence independently.

The Core Question: "To get $DP[i][j]$, what do we do with $A[i]$ and $B[j]$?"

Archetype 1: Related Problems

Problems following this pattern:

- **Edit Distance:**
  $DP[i][j] = \min$ operations to transform $A[1..i] \to B[1..j]$

  Decision: "Match? Replace? Insert? Delete?"

  Recurrence: $ED[i,j] = \min \begin{cases} ED[i-1,j]+1 \\ ED[i,j-1]+1 \\ ED[i-1,j-1]+[A[i] \neq B[j]] \end{cases}$

- **Longest Common Substring (contiguous):**
  $DP[i][j] =$ length of longest common substring *ending* at i, j

  Decision: "Does $A[i] = B[j]$? If yes, extend; if no, reset to 0."

Pattern Recognition: Two sequences + element-by-element decisions $\implies$ $DP[i][j]$ for prefixes.

2) Sequential Decision

Archetype 2: Sequential Decision Problems

Core Pattern

Process elements **one at a time** in a sequence. Make decisions based on previous elements with order-dependent rules.

Key Insight: This archetype splits into two subpatterns depending on whether the solution requires **contiguous** or **non-contiguous** elements:

1. **Contiguous/Local Pattern:** Solution requires adjacent/recent elements only
   - Can often decide using only $DP[i-1]$ or a small local window
   - Leads to $O(n)$ or $O(n \log n)$ solutions
2. **Non-Contiguous/Global Pattern:** Solution can skip elements freely
   - Must consider *all* valid predecessors $j < i$
   - Naive approach: $O(n^2)$, but often optimizable!

Archetype 2a: Contiguous/Local Pattern

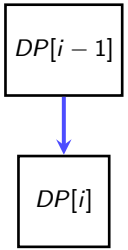
Canonical Example: Maximum Subarray Sum

- **Subproblem:** $DP[i]$ = max sum of subarray ending at i
- **Key Feature:** Subarrays must be **contiguous**
- **Decision:** Only two choices at i:
  1. Extend previous subarray: $DP[i-1] + A[i]$
  2. Start fresh at i: $A[i]$
- **Recurrence:**

$$DP[i] = \max(DP[i-1] + A[i], A[i])$$

- **Runtime:** $O(n)$ — only need previous element!

Only depends on **previous** element



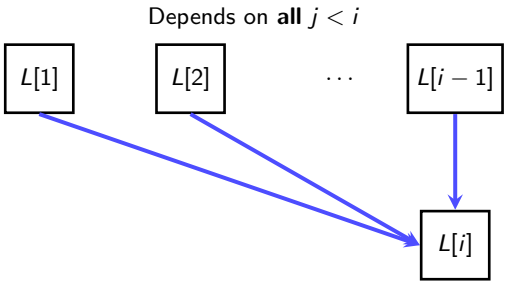
Why Local? The contiguity constraint means we can only extend the *immediately previous* subarray or start over.

Archetype 2b: Non-Contiguous/Global Pattern (Naive)

Canonical Example: Longest Ascending Subsequence (LAS)

- **Subproblem:** $L[i]$ = length of LAS ending at i
- **Key Feature:** Subsequences can **skip** elements
- **Decision:** Must check *all* valid predecessors:
  - Which $j < i$ has $A[j] < A[i]$?
  - Which gives best $L[j]$?
- **Recurrence:**

$$L[i] = 1 + \max(\{L[j] \mid j < i, A[j] < A[i]\} \cup \{0\})$$



- **Runtime:** $O(n^2)$ — must check all $j < i$

Why Global? We can skip elements freely, so must consider *all* possible predecessors that satisfy $A[j] < A[i]$.

Archetype 2b: Optimizing Non-Contiguous Problems

The $O(n^2)$ solution is often just the starting point!

We have seen two problems where we can define the subproblem in a way as to narrow down the viable candidates and make them efficiently searchable.

1. **Jump Game:**
   - Instead of $DP[i]$ = "min jumps to reach i" (requires checking all $j < i$)
   - Use $DP[k]$ = "farthest position reachable in k jumps"
   - Only process each position once $\implies$ $O(n)$

2. **Longest Ascending Subsequence:**
   - The naive approach checks all $j < i$ where $A[j] < A[i]$
   - Key insight: We only care about the *best* (longest) subsequence ending with each possible value
   - Use binary search on (smallest possible) tail values $\implies$ $O(n \log n)$
   - Maintains: "smallest tail value for each subsequence length"

The archetype helps you find the $O(n^2)$ solution; optimization requires deeper problem-specific insight!

Archetype 2: Related Problems by Subpattern

Contiguous/Local ($O(n)$):

- **Max Subarray Sum:**
  $DP[i] = \max(DP[i-1] + A[i], A[i])$
- **Min-Cost Stairs (last week):**
  $DP[i] =$
  $\min(DP[i-1]+c[i-1], DP[i-2]+c[i-2])$
  jump from one or two steps down at cost c_i

Non-Contiguous/Global:

- **LAS (naive):** $O(n^2)$
  $DP[i] = 1 + \max\{DP[j] \mid A[j] < A[i]\}$
  Optimized: $O(n \log n)$ with binary search
- **Jump Game (naive):** $O(n^2)$
  $DP[i] = 1 + \min\{DP[j] \mid j + A[j] \geq i\}$
  Optimized: $O(n)$ with subproblem change
- **Max Sum Asc. Subseq.:** $O(n^2)$
  Similar structure to LAS

Pattern Recognition: Contiguous → local dependency; Non-contiguous → global dependency (but often optimizable).

Deep Dive: Longest Ascending Subsequence (LAS)

Definition 2.11 (LAS)

Given an array $A[1..n]$. We seek the **length** of a longest subsequence of A whose elements are ascending.

Example: $A = (2, 13, 17, 9, 11, 4, 78, 28, 15, 25, 99)$

- Ascending subsequence: $(2, 13, 17, 78, 99) \implies$ Length 5
- Longest ascending subsequence (LAS): $(2, 9, 11, 15, 25, 99) \implies$ Length 6

LAS: Attempt 1 (Fails)

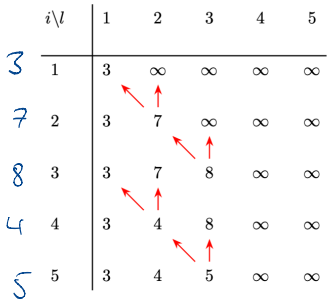
- **Subproblem:** $LAT(i)$ = Length of the LAS in array $A[1..i]$.
- **Problem:** The solution for $A[1..i-1]$ doesn't necessarily help us find the solution for $A[1..i]$.
  Example: $A = (1, 2, 5, 3, 4)$
  - $LAT(3)$ for $(1, 2, 5)$ is 3. (Sequence: $(1, 2, 5)$)
  - $LAT(4)$ for $(1, 2, 5, 3)$ is 3. (Sequence: $(1, 2, 5)$ or $(1, 2, 3)$)
  - $LAT(5)$ for $(1, 2, 5, 3, 4)$ is 4. (Sequence: $(1, 2, 3, 4)$)

  We cannot simply compute $LAT(5)$ from $LAT(4)$!
- **Realization:** We need to remember more than just the length. The **ending values** of the subsequences are important.

LAS: Better DP ($O(n^2)$)

- **Idea:** For each possible length l, we store the **smallest possible ending element** of an AS of that length.
- **Subproblem:** $DP[i][l] :=$ the smallest possible ending element of an AS of length l in the range $A[1..i]$.
- **Recurrence:** To compute $DP[i][l]$, we consider $A[i]$:
  1. **We don't use** $A[i]$, instead use $DP[i-1][l]$.
  2. **We use** $A[i]$**:** We can append $A[i]$ to an AS of length $l-1$ ending in $DP[i-1][l-1]$.

  $$DP[i][l] = \min\{DP[i-1][l]),\quad A[i] \text{ (if } A[i] > DP[i-1][l-1])\}$$
- **Runtime:** We fill an $n \times n$ table. Each cell takes $O(1)$.
  $\implies$ $\mathbf{O(n^2)}$.



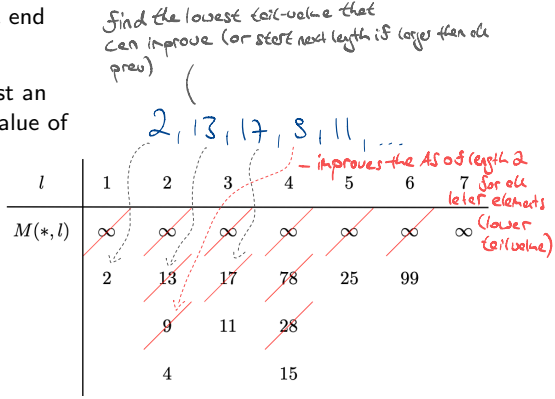
2.5 Beispiel der Tabelle $M(i, l)$ mit $A = [3, 7, 8, 4, 5]$

LAS: The $O(n \log n)$ Solution

- **Observation 1:** The rows of the $DP[i][...]$ table are sorted: $DP[i][l] < DP[i][l+1]$ (since $DP[i][l]$ is the **smallest** end value).
- **Algorithm:** We don't need the whole $O(n^2)$ table, just an array $DP[1..n]$, where $DP[l]$ stores the smallest end value of an AS of length l found *so far*.

  $T[1..n] \leftarrow \infty$, $T[0] \leftarrow -\infty$
  for $i \leftarrow 1$ to n **do**
  Find l (with binary search) such that $T[l-1] < A[i] \leq T[l]$
  $T[l] \leftarrow A[i]$
  end for
  return largest l such that $T[l] < \infty$

- **Runtime:** n iterations. Each iteration: one binary search ($O(\log n)$). $\implies$ $\mathbf{O(n \log n)}$.

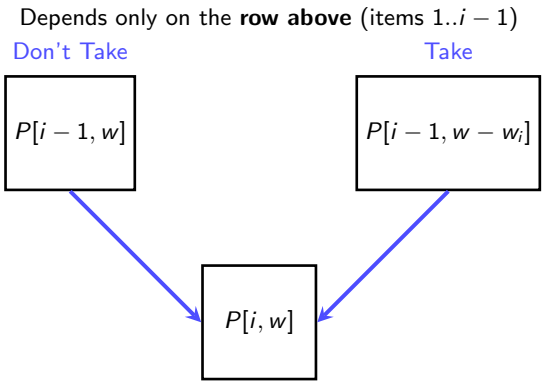


3) Bounded Resources

Archetype 3: Bounded Resource Problems

- **Archetype:** 0/1 Knapsack
- **Key Feature: Resource Management**. We have a limit and must make binary choices. Order doesn't matter.
- **Subproblem:** $P[i][w]$ = Max profit using items $1..i$ with capacity w.
- **Key Idea:** For item i, make a single binary choice:
  1. **Don't Take:** $P[i-1][w]$
  2. **Take:** $p_i + P[i-1][w-w_i]$
- **Recurrence:**

$$P[i,w] = \max(P[i-1,w],\ p_i + P[i-1,w-w_i])$$



Archetype 3: Why 2D?

Intuition for the Subproblem:

- We need to track two things:
  1. Which items have we *considered*? (the i in $DP[i][k]$)
  2. How much resource have we *used*? (the k in $DP[i][k]$)

Why 1D Fails for Knapsack

Try: $P[i]$ = max profit using items $1..i$

Problem: We don't know **how much capacity was used**!

If we took high-profit items early, we might have no capacity left. But maybe skipping those items leaves room for a better combination later.

We **need to track remaining capacity** at each step.

The Core Question: "For item i, do we 'Take It' or 'Leave It'?"

Problem: Subset Sum Problem

Definition 2.4 (Subset Sum)

Given: n natural numbers $A[1], ..., A[n]$ and a target number b.

Wanted: A subset $I \subseteq \{1, ..., n\}$ such that $\sum_{i \in I} A[i] = b$.

Example:

$A = (5, 3, 10, 7, 3, 1)$, $b = 9$.

- $I = \{1, 5, 6\} \implies A[1] + A[5] + A[6] = 5 + 3 + 1 = 9$. **Answer: Yes.**

$A = (5, 3, 10, 7, 3, 1)$, $b = 2$.

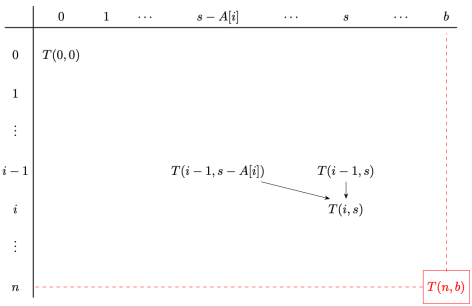
- No subset sums to 2. **Answer: No.**

Naive Approach: Try all 2^n subsets $\implies$ Exponential!

Subset Sum: The DP Solution

- **Subproblem:** $T(i, s) :=$ (boolean) "Is it possible to achieve the sum s with a subset of $A[1..i]$?"
- **Recurrence:** To compute $T(i, s)$, we have two choices for the element $A[i]$:
  1. **We do not use $A[i]$:** We must have already achieved sum s with $A[1..i-1]$. $\implies$ $T(i-1, s)$
  2. **We use $A[i]$:** We must have achieved the remaining sum $s - A[i]$ with $A[1..i-1]$. $\implies$ $T(i-1, s-A[i])$

  $T(i, s) = T(i-1, s) \quad \vee \quad T(i-1, s-A[i])$
- **Base Cases:**
  - $T(0, 0) = 1$ (Sum 0 is possible with 0 elements).
  - $T(0, s) = 0$ for $s > 0$.
- **Solution:** $T(n, b)$.



Subset Sum: Analysis

Runtime

- **Computation:** Fill an $(n+1) \times (b+1)$ table, row by row.

  Each cell $T(i, s)$ only depends on the previous row $(i-1)$.

  Each cell takes $O(1)$ time.

  $\implies O(n \cdot b)$

But notice that the **length of the input for value** b is $\log b$!

Pseudo-polynomial Runtime

- The runtime is polynomial in n and the *value* b.
- It is **not** polynomial in the *input length* (which is $n + \log b$ bits).
- If $b = 2^n$, the runtime is $\Theta(n \cdot 2^n) \implies$ exponential!

Problem: Knapsack Problem (0/1 Knapsack)

Definition 2.6 (Knapsack)

Given: n items, each with weight w_i and profit p_i. A knapsack with capacity W.

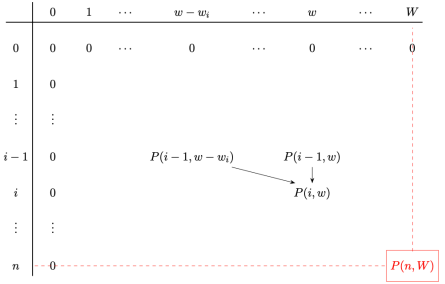
Wanted: A subset $I \subseteq \{1, ..., n\}$ such that $\sum_{i \in I} w_i \leq W$ and $\sum_{i \in I} p_i$ is **maximal**.

Again: Greedy strategies (e.g., "highest profit first", "best p_i/w_i ratio") **do not work** for the 0/1 Knapsack problem.

Knapsack: DP Solution 1 ($O(nW)$)

- **Idea:** Very similar to Subset Sum, but instead of 'Yes/No', we ask for the optimal sol.
- **Subproblem:** $DP[i][w]$:= the **maximum profit** one can achieve with a subset of $A[1..i]$ given a weight limit w.
- **Recurrence:** For $DP[i][w]$, two choices for $A[i]$:
  1. **We do not take $A[i]$:**
     Maximum profit is $DP[i-1][w]$.
  2. **We take $A[i]$**
     and get profit: $p_i + DP[i-1][w-w_i]$.

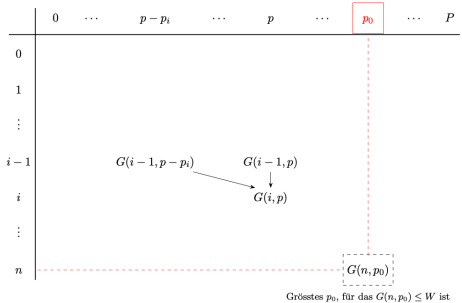
  $DP[i][w] = \max\{DP[i-1][w], \quad p_i + DP[i-1][w-w_i]\}$
- **Base Case:** $DP[0][w] = 0$ for all w.
- **Runtime:** $O(n \cdot W)$ (**pseudo-polynomial**).



Knapsack: DP Solution 2 ($O(nP)$)

- **Idea:** Swap the roles of weight and profit.
- **Subproblem:** $G(i, p) :=$ the **minimum weight** needed to achieve *exactly* the profit p with a subset of $A[1..i]$. ($DP[i][p] = \infty$ if p is not reachable).
- **Recurrence:** For $DP[i][p]$, two choices for $A[i]$:
  1. **We do not take $A[i]$:**
     Min. weight is $DP[i-1][p]$.
  2. **We take $A[i]$ (if $p_i \leq p$):**
     Weight is w_i plus the min. weight to get the remaining profit $p - p_i$ with $A[1..i-1]$: $w_i + DP[i-1][p-p_i]$.

  $$DP[i][p] = \min\{DP[i-1][p], \quad w_i + DP[i-1][p-p_i]\}$$
- **Base Cases:** $DP[0][0] = 0$, $DP[0][p] = \infty$ for $p > 0$.
- **Solution:** Find $\max\{p_0 \mid DP[n][p_0] \leq W\}$.
- **Runtime:** $O(n \cdot P)$ (where $P = \sum p_i$).



The Problem with Knapsack

Subset Sum and Knapsack are **NP-complete**.

This means we don't expect a *truly* polynomial algorithm (unless P=NP (don't try to prove it...)).

- Our $O(nW)$ and $O(nP)$ solutions are **pseudo-polynomial**.
- If W **and** P are huge? (e.g., $W, P > 2^n$), both DP algorithms would be exponential!
- **New Idea:** What if an *almost* optimal solution is good enough?
- **Goal:** A $(1 - \epsilon)$-approximation algorithm that runs in time poly$(n, 1/\epsilon)$.

FPTAS: The Idea

The $O(nP)$ algorithm is slow because P (the maximum profit) can be large.

The Trick: Round the profits!

If we divide all profits by K and round, the new maximum profit $P' \approx P/K$.

$\implies$ The $O(nP')$ algorithm is then $O(nP/K)$ fast.

- **Trade-off:**
  - Large K: Fast runtime, but inaccurate solution (lots of rounding error).
  - Small K: Slow runtime, but accurate solution.
- **Goal:** Choose K "just right" so that the error is $\leq \epsilon \cdot P_{OPT}$ and the runtime remains poly$(n, 1/\epsilon)$.

FPTAS: Runtime

Let $\epsilon > 0$ be our desired approximation error (e.g., $\epsilon = 0.1$).

1. Find $p_{max} = \max_i\{p_i\}$.
2. Set the scaling factor $K := \frac{\epsilon \cdot p_{max}}{n}$.
3. Create new, scaled profits: $\overline{p}_i := K \cdot \lfloor \frac{p_i}{K} \rfloor$.
4. Solve the Knapsack problem with weights w_i and profits $\overline{p}_i$.
5. Return the found subset $\overline{OPT}$ as the solution to the original problem.

Runtime

The runtime is $O(\frac{n \cdot \overline{P}}{K})$, since we only need to fill-in every k-th column in the DP-table.

Then since $\overline{P} = \sum_{i=1}^{n} \overline{p_i} \leq n \cdot p_{max}$, the runtime is:

$O(\frac{n\overline{P}}{K}) \leq O(\frac{n(n \cdot p_{max})}{K}) = O(\frac{n^2 p_{max}}{\epsilon p_{max}/n}) = O(n^3/\epsilon)$

This is a **F**ully **P**olynomial **T**ime **A**pproximation **S**cheme (FPTAS).

FPTAS: The Algorithm

Let $\epsilon > 0$ be our desired approximation error (e.g., $\epsilon = 0.1$).

Quality

The solution $\overline{OPT}$ has a profit $p(\overline{OPT}) \geq (1 - \epsilon) \cdot p(OPT)$.

Let $\overline{p_i} = K \cdot \left\lfloor \frac{p_i}{K} \right\rfloor$, $\overline{OPT}$ = approx. solution, OPT = optimal sol.

Notice: $p_i - K \overset{(1)}{\leq} \overline{p_i} \overset{(2)}{\leq} p_i \overset{(3)}{\leq} p_{max} \overset{(4)}{\leq} P_{OPT}$

(1) $\overline{p_i} = \left\lfloor \frac{p_i}{K} \right\rfloor \cdot K$, i.e., rounded down to the next multiple of K. Hence $p_i - \overline{p_i} \leq K$ (rounding can cause a diff of at most K.

It follows:

($\overline{OPT}$ is the opt. set for $\overline{p_i}$) ($|OPT| \leq n$) (def. P_{OPT}) ($\frac{P_{OPT}}{p_{max}} \geq 1$ by (4)) (factor out P_{OPT})

$$\sum_{i \in \overline{OPT}} \overline{p_i} \geq \sum_{i \in OPT} \overline{p_i} \overset{(1)}{\geq} \sum_{i \in OPT} (p_i - K) \geq \sum_{i \in OPT} p_i - n \cdot K = P_{OPT} - n \cdot K \geq P_{OPT} - n \cdot K \cdot \frac{P_{OPT}}{p_{max}} = P_{OPT}\left(1 - \frac{nK}{p_{max}}\right)$$

different sets?

Again, letting $K = \frac{\epsilon \cdot p_{max}}{n}$, we obtain: $\sum_{i \in \overline{OPT}} \overline{p_i} = P_{\overline{OPT}} \geq P_{OPT}(1 - \epsilon)$

Archetype 3: Related Problems

Problems following this pattern:

- **Subset Sum:**
  $DP[i][s]$ = "can we make sum s using items $1..i$?" (boolean)
  Decision: "Include $A[i]$ or not?"
  Recurrence: $T[i, s] = T[i-1, s] \vee T[i-1, s - A[i]]$
- **Coin Change:**
  $DP[i][v]$ = min coins to make value v using first i coin types
  Decision: "How many of coin i do I use?"
- **Partition Problem:**
  $DP[i][s]$ = "can we partition items $1..i$ to sum s?"

Pattern Recognition: Items + resource constraint $\implies$ $DP[i, \text{resource}]$.

Summary

Summary: DP Patterns

Archetype	Key Feature	Subproblem $DP[\dots]$	Dependency
Sequential Decision (Order matters)	Valid Extension (Kadane, LAS)	$DP[i]$ = Sol. *ending at* i	Local: $DP[i-1]$ Global: **All** $j < i$
Sequence Alignment (2 sequences)	Alignment (LCS, Edit Dist.)	$DP[i][j]$ = Sol. for prefixes $A[1..i]$, $B[1..j]$	$DP[i-1][j]$, $DP[i][j-1]$, $DP[i-1][j-1]$
Bounded Resource (Subset selection)	Resource Mgmt. (Knapsack, Subset Sum)	$DP[i][k]$ = Sol. using items $1..i$, resource k	$DP[i-1][k]$, $DP[i-1][k-\text{cost}_i]$

The Universal Question:

"What information do I need about smaller subproblems to make an optimal decision for the current problem?"

Your subproblem parameters should capture **exactly** that information.

Additional Practice

DP - Partition Problem (1/3)

Description:

Given a non-empty array of positive integers A, determine if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

Example:

- Input: $A = [1, 5, 11, 5]$
- Output: true
- Explanation: The array can be partitioned as 1, 5, 5 (sum=11) and 11 (sum=11).
- Input: $A = [1, 2, 3, 5]$
- Output: false
- Explanation: The total sum is 11 (odd), so it's impossible to partition into two equal halves.

DP - Partition Problem (2/3)

Compute the solution using bottom-up dynamic programming and state the run time of your algorithm.

Address the following aspects in your solution:

1. **Definition of the DP table:** What are the dimensions of the table DP [. . .]? What is the meaning of each entry?

2. **Computation of an entry:** How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.

see next page for cont.

DP - Partition Problem (3/3)

3. **Calculation order:** In which order can entries be computed so that values needed for each entry have been determined in previous steps?

4. **Extracting the solution:** How can the final solution be extracted once the table has been filled?

5. **Runtime:** What is the run time of your solution?

When you're done, try implementing your solution on CodeExpert!

Peer Grading

Peer-Grading Exercise

This week's peer-grading exercise is **Exercise 6.3**

Please follow the usual process:

- Grade the assigned group's submission.
- Give constructive feedback and upload your feedback to Moodle by 23.59 tonight, at the latest.
- Contact me if you don't receive a submission to grade.