

12

Dynamic Programming

21 January 2024

Prof. Dr. Sebastian Wild

Learning Outcomes

Unit 12: *Dynamic Programming*

1. Be able to apply the DP paradigm to solve new problems.

12 Dynamic Programming

- 12.1 Elements of Dynamic Programming
- 12.2 DP & Matrix Chain Multiplication
- 12.3 Greedy as Special Case of DP
- 12.4 The Bellman-Ford Algorithm
- 12.5 Making Change in Pre-1971 UK
- 12.6 Optimal Merge Trees & Optimal BSTs
- 12.7 Edit Distance

12.1 Elements of Dynamic Programming

Introduction

applicable to many problems

- ▶ **Dynamic Programming (DP)** is a powerful algorithm **design pattern** for exact solutions to **optimization** problems

- ▶ Some commonalities with Greedy Algorithms, but with an element of brute force added in

DP = “careful brute force” (Erik Demaine)

- ▶ often yields polynomial time, but usually not linear time algorithms
- ▶ for many problems the *only* way we know to build efficient algorithms
- ▶ **Naming fun:** The term “dynamic programming”, due to Richard Bellman from around 1953, does not refer to computer programming; rather to a program (= plan, schedule) changing with time. It seems to have been at least partly marketing babble devoid of technical meaning . . .

Plan of the Unit


1. Abstract steps of DP (briefly)
2. Details on a concrete example (*matrix chain multiplication*)
3. More examples!

The 6 Steps of Dynamic Programming

1. Define **subproblems** (and relate to original problem)
2. **Guess** (part of solution) \rightsquigarrow local brute force
3. Set up **DP recurrence** (for quality of solution)
4. Recursive implementation with **Memoization**
5. Bottom-up **table filling** (topological sort of subproblem dependency graph)
6. **Backtracing** to reconstruct optimal solution

► Steps 1–3 require insight / creativity / intuition;
Steps 4–6 are mostly automatic / same each time

\rightsquigarrow Correctness proof usually at level of DP recurrence

 running time too! worst case time = #subproblems \cdot time to find single best guess

When does DP (not) help?

► No Silver Bullet

DP is the most widely applicable design technique, but can't *always* be applied

1. Vitally important for DP to be correct:

Bellman's Optimality Criterion

**For a *correctly guessed* fixed part of the solution,
any optimal solution to the corresponding subproblems
must yield an *optimal solution* to the overall problem (once combined).**

2. Also, the total **number of different subproblems** should be "*small*"

(DP potentially still works correctly otherwise, but won't be *efficient*.)

at most polynomial in n



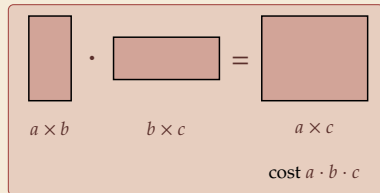
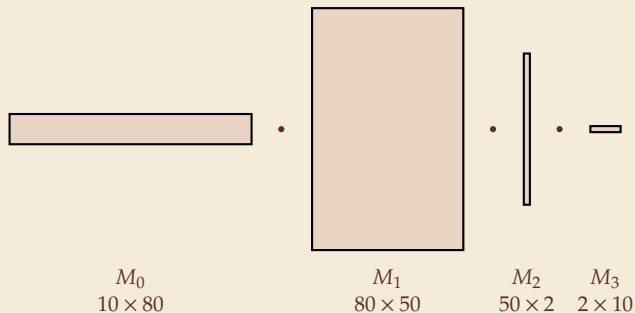
12.2 DP & Matrix Chain Multiplication

The Matrix-Chain Multiplication Problem

Consider the following exemplary problem

- ▶ We have a product $M_0 \cdot M_1 \cdot \dots \cdot M_{n-1}$ of n matrices to compute
- ▶ Since (matrix) multiplication is associative, it can be evaluated in different orders.
- ▶ For non-square matrices of different sizes, different order can change costs dramatically
 - ▶ Assume elementary matrix multiplication algorithm:
 - ↪ Multiplying $a \times b$ -matrix with $b \times c$ matrix costs $a \cdot b \cdot c$ integer multiplications
- ▶ **Given:** Row and column counts $c[0..n]$ and $r[0..n]$ with $r[i+1] = c[i]$ for $i \in [0..n-1]$
(corresponding to matrices M_0, \dots, M_{n-1} with $M_i \in \mathbb{R}^{r[i] \times c[i]}$)
- ▶ **Goal:** parenthesization of the product chain with minimal cost
 - really a binary tree with n leaves!

Matrix-Chain Multiplication – Example



Parenthesization	Cost (integer multiplications)		
$M_0 \cdot (M_1 \cdot (M_2 \cdot M_3))$	$1000 + 40\,000 + 8000$	$=$	$49\,000$
$M_0 \cdot ((M_1 \cdot M_2) \cdot M_3)$	$8000 + 1600 + 8000$	$=$	$17\,600$
$(M_0 \cdot M_1) \cdot (M_2 \cdot M_3)$	$40\,000 + 1000 + 5000$	$=$	$46\,000$
$(M_0 \cdot (M_1 \cdot M_2)) \cdot M_3$	$8000 + 1600 + 200$	$=$	$9\,800$
$((M_0 \cdot M_1) \cdot M_2) \cdot M_3$	$40\,000 + 1000 + 200$	$=$	$41\,200$

first or last operation
Greedy fails both ways!

Matrix-Chain Multiplication – How about Brute Force?

If Greedy doesn't give optimal parenthesization, maybe just try all?

- ▶ parenthesizations for n matrices = binary trees with n leaves (*evaluation trees*)
= binary trees with $n - 1$ (internal) nodes

- ▶ How many such trees are there?

- ▶ Let's write $m = n - 1$;

- ▶ $C_0 = 1, C_1 = 1, C_2 = 2, C_3 = 5$

- ▶ $C_m = \sum_{r=1}^m C_{r-1} \cdot C_{m-r} \quad (m \geq 1)$

generating functions / combinatorics / guess (OEIS!) & check ...

- ▶ Can show $C_n = \frac{1}{n+1} \binom{2n}{n} \sim \frac{1}{\sqrt{\pi}} \cdot \frac{4^n}{n^{3/2}}$

\rightsquigarrow exponentially many trees (almost 4^n)

$C_{20} = 6\,564\,120\,420, C_{30} = 3\,814\,986\,502\,092\,304$

\rightsquigarrow A brute-force approach is utterly hopeless

\rightsquigarrow Dynamic programming to the rescue!

Matrix-Chain Multiplication – Step 1: Subproblems

- ▶ Key ingredient for DP: Problem allows for recursive formulation
Need to decide:
 1. What are the **subproblems** to consider?
 2. How can the **original problem** be expressed as subproblem(s)?
- ▶ Often requires to solve a more general version of the problem

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

Here:

1. **Subproblems** = Ranges of matrices $[i..j)$ $0 \leq i \leq j \leq n$
i. e., optimal parenthesization for each range $M_i, M_{i+1}, \dots, M_{j-1}$
 2. **Original problem** = range $[0..n)$
- ▶ **Intuition:**
 - ▶ Any subtree in binary multiplication tree covers some range $[i..j)$
(matrix multiplication is not commutative \rightsquigarrow left-right order has to stay)
 - ▶ left and right factors of a multiplication don't "see/influence" each other

Matrix-Chain Multiplication – Step 2: Guess

- ▶ Usually, any subproblem can be split into smaller subproblems in different ways
- ▶ Which way to decompose gives best solution not known *a priori*
- ↪ Assuming we can correctly *guess* this part; how to solve problem?

▶ Here: **Guess** last multiplication / root of binary tree

↪ index $k \in [i + 1 .. j)$ so that $[i..j)$ computed with **last** multiplication
 $(M_i \cdot \dots \cdot M_{k-1}) \cdot (M_k \cdot \dots \cdot M_{j-1})$

↪ optimal parenthesization of M_i, \dots, M_{k-1} and M_k, \dots, M_{j-1} computed recursively
(corresponds to subproblems $[i..k)$ and $[k..j)$)

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

Matrix-Chain Multiplication – Step 3: DP Recurrence

- ▶ With subproblems and guessed part fixed, we try to express total **value/cost of solution** *recursively*

↪ *We ignore the actual solution and just compute its cost!*

- ▶ Often good to prove correctness at level of recurrence

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

- ▶ Here: **Recurrence** for $m(i, j)$ = total number of integer multiplications used in best parenthesization of $[i..j]$

↪ Set up recurrence, including any base cases.

$$m(i, j) = \begin{cases} 0 & \text{if } j - i \leq 1 \\ \min \left\{ \overbrace{m[i, k] + m[k, j]}^{\text{recursive cost}} + \overbrace{r[i] \cdot r[k] \cdot c[j-1]}^{\text{cost of last multiplication}} : k \in [i+1..j] \right\} & \text{otherwise} \end{cases}$$

↗
best k chosen by *local brute force*

Matrix-Chain Multiplication – Correctness

Claim: Let $m(i, j)$ for $0 \leq i \leq j \leq n$ be defined by the recurrence

$$m(i, j) = \begin{cases} 0 & \text{if } j - i \leq 1 \\ \min\{m(i, k) + m(k, j) + r[i] \cdot r[k] \cdot c[j - 1] : k \in [i + 1 .. j]\} & \text{otherwise} \end{cases}$$

Then $m(i, j) = \text{\#integer multiplications in best parenthesization of } M_i \cdots M_{j-1}.$

Proof:

- ▶ **IB:** When $j - i \leq 1$ we have an empty product ($j = i$) or a single matrix ($j = i + 1$)
In both cases, no multiplications are needed and $m(i, j) = 0$.
 - ▶ **IS:** Given $j - i \leq 2$ matrices and an optimal evaluation tree T for them.
 - ▶ T 's root must be a last product of left and right subterms $(M_i \cdots M_{k-1}) \cdot (M_k \cdots M_{j-1})$ for some $i < k < j$, with cost $r[i]r[k]c[j - 1]$.
 - ▶ Moreover, left and right subtree T_ℓ and T_r of the root must be optimal evaluation trees for subproblems $[i..k)$ and $[k..j)$; (otherwise can improve T)
- ↪ By IH, the cost of T_ℓ and T_r are given by $m(i, k)$ and $m(k, j)$
- ↪ $m(i, j) = \text{cost of } T$

Matrix-Chain Multiplication – Step 4: Memoization

- ▶ Write **recursive** function to compute recurrence
- ▶ But *memoize* all results! (symbol table: subproblem \mapsto optimal cost)

\rightsquigarrow First action of function: check if subproblem known

- ▶ If so, return cached optimal cost
- ▶ Otherwise, compute optimal cost and remember it!

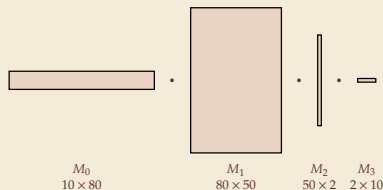
1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

```
1 procedure totalMults( $r[i..j]$ ,  $c[i..j]$ ):
2   if  $j - i \leq 1$ 
3     return 0
4   else
5      $best := +\infty$ 
6     for  $k := i + 1, \dots, j - 1$ 
7        $m_l := \text{cachedTotalMults}(r[i..k], c[i..k])$ 
8        $m_r := \text{cachedTotalMults}(r[k..j], c[k..j])$ 
9        $m := m_l + m_r + r[i] \cdot r[k] \cdot c[j - 1]$ 
10       $best := \min\{best, m\}$ 
11   end for
12   return  $best$ 
```

$$m(i, j) = \begin{cases} 0 & \text{if } j - i \leq 1 \\ \min\{m(i, k) + m(k, j) + r[i] \cdot r[k] \cdot c[j - 1] : k \in [i + 1 .. j]\} & \text{otherwise} \end{cases}$$

```
13 procedure cachedTotalMults( $r[i..j]$ ,  $c[i..j]$ ):
14   //  $m[0..n][0..n]$  initialized to NULL at start
15   if  $m[i][j] == \text{NULL}$ 
16      $m[i][j] := \text{totalMults}(r[i..j], c[i..j])$ 
17   return  $m[i, j]$ 
```

Matrix-Chain Multiplication – Example Memoization



$n = 4$

$r[0..n) = [10, 80, 50, 2]$

$c[0..n) = [80, 50, 2, 10]$

$i \backslash j$		0	1	2	3	4
$m[i][j]$	0	0	0	40000	9600	9800
	1	—	0	0	8000	9600
	2	—	—	0	0	1000
	3	—	—	—	0	0
	4	—	—	—	—	0

Matrix-Chain Multiplication – Runtime Analyses

```
1 procedure totalMults( $r[i..j]$ ,  $c[i..j]$ ):  
2   if  $j - i \leq 1$   
3     return 0  
4   else  
5      $best := +\infty$   
6     for  $k := i + 1, \dots, j - 1$   
7        $m_l := \text{cachedTotalMults}(r[i..k], c[i..k])$   
8        $m_r := \text{cachedTotalMults}(r[k..j], c[k..j])$   
9        $m := m_l + m_r + r[i] \cdot r[k] \cdot c[j - 1]$   
10       $best := \min\{best, m\}$   
11   end for  
12   return  $best$ 
```

\rightsquigarrow total running time $\Theta(n^3)$

```
13 procedure cachedTotalMults( $r[i..j]$ ,  $c[i..j]$ ):  
14   //  $m[0..n][0..n]$  initialized to NULL at start  
15   if  $m[i][j] == \text{NULL}$   
16      $m[i][j] := \text{totalMults}(r[i..j], c[i..j])$   
17   return  $m[i, j]$ 
```

- ▶ With memoization, compute each subproblem at most once
- ▶ nonrecursive cost (totalMults):
 $O(j - i) = O(n)$
- ▶ Number of subproblems $[i..j]$ for
 $0 \leq i \leq j \leq n$

$$\sum_{0 \leq i \leq j \leq n} 1 = \sum_{i=0}^n \sum_{j=i}^n 1 = \Theta(n^2)$$

Matrix-Chain Multiplication – Step 5: Table Filling

- ▶ Recurrence induces a DAG on subproblems (who calls whom)
 - ▶ Memoized recurrence traverses this DAG (DFS!)
 - ▶ We can slightly improve performance by systematically computing subproblems following a fixed topological order
- ▶ **Topological order** here: by **increasing length** $\ell = j - i$, then by i

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

```
1 procedure totalMultsBottomUp( $r[0..n]$ ,  $c[0..n]$ ):
2    $m[0..n][0..n] := 0$  // initialize to 0
3   for  $\ell = 2, 3, \dots, n$  // iterate over subproblems ...
4     for  $i = 0, 1, \dots, n - \ell$  // ... in topological order
5        $j := i + \ell$ 
6        $m[i][j] := +\infty$ 
7       for  $k := i + 1, \dots, j - 1$ 
8          $q := m[i][k] + m[k][j] + r[i] \cdot r[k] \cdot c[j - 1]$ 
9          $m[i][j] := \min\{m[i][j], q\}$ 
10  return  $m[0..n][0..n]$ 
```

- ▶ Same Θ -class as memoized recursive function
- ▶ In practice usually substantially faster
 - ▶ lower overhead
 - ▶ predictable memory accesses

Matrix-Chain Multiplication – Step 6: Backtracing

- ▶ So far, only determine the **cost** of an optimal solution
 - ▶ But we also want the solution itself
- ▶ By *retracing* our steps, we can determine/construct one!
- ▶ Here: output a parenthesized term recursively

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

```
1 procedure matrixChainMult(r[0..n], c[0..n]):
2   m[0..n][0..n] := totalMultsBottomUp(r[0..n], c[0..n])
3   return traceback([0..n])
4
5 procedure traceback([i..j]):
6   if j - i == 1
7     return Mi
8   else
9     for k := i + 1, ..., j - 1
10      q := m[i][k] + m[k][j] + r[i] · r[k] · c[j - 1]
11      if m[i][j] == q
12        return (traceback([i..k])) · (traceback([k..j]))
13    end for
14  end if
```

- ▶ follow recurrence a second time
- ▶ always have for running time:
backtracing = $O(\text{computing } M)$
 - ↪ computing optimal cost and
computing optimal solution have
same complexity
- ▶ speedup possible by
remembering correct guess k for
each subproblem

Summary: The 6 Steps of Dynamic Programming

1. Define **subproblems** and how **original problem** is solved

2. What part of solution to **guess**?

3. Set up **DP recurrence** for quality/cost of solution

↪ Prove **correctness** here: induction over subproblems following recurrence

↪ Analyze running **time complexity** here: $\# \text{subproblems} \cdot \text{non-recursive time}$

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

— (Basically) cookie-cutter approach from here on —



4. Recursive implementation with **Memoization**: mutually recursive functions with cache
or

5. Bottom-up **table filling**: define topological order of subproblem dependency graph

6. **Backtracing** to reconstruct optimal solution: Recursively retrace cost recurrence

12.3 Greedy as Special Case of DP

Dynamic Greedy

- ▶ Every Greedy Algorithm can also be seen as a DP algorithm **without guessing**
- ↪ For new problems, it can help to first follow the DP roadmap and then check if we can select the “correct” guess without local brute force
- ▶ If so, we then recurse on a single branch of subproblems
- ↪ Greedy Algorithm doesn't need memoization or bottom-up table filling, but can do direct recursion instead

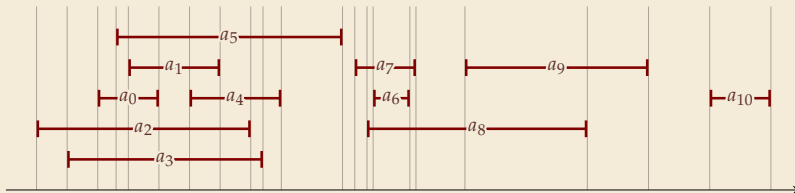
Recall Unit 11

The Activity selection problem

- **Activity Selection:** scheduling for *single* machine, jobs with *fixed* start and end times pick a *subset* of jobs without *conflicts*

Formally:

- **Given:** Activities $A = \{a_0, \dots, a_{n-1}\}$, each with a start time s_i and finish time f_i ($0 \leq s_i < f_i < \infty$)
- **Goal:** Subset $I \subseteq [0..n)$ of tasks such that $i, j \in I \wedge i \neq j \implies [s_i, f_i) \cap [s_j, f_j) = \emptyset$ and $|I|$ is maximal among all such subsets
- We further assume that jobs are sorted by finish time, i.e., $f_0 \leq f_1 \leq \dots \leq f_{n-1}$ (if not, easy to sort them in $O(n \log n)$ time)



DP Algorithm for Activity Selection

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

1. **Subproblems:** $A_{i,j} = \{a_\ell \in A : s_\ell \geq f_i \wedge f_\ell \leq s_j\}$
(after a_i finishes and before a_j begins)

Original problem: $A_{-1,n}$ with dummy tasks $s_{-1} = -\infty, f_n = +\infty$

2. **Guess:** Task $k \in I^*$

3. **DP Recurrence:** Denote $c(i, j) = |I^*(A_{i,j})|$ = maximum #independent tasks in $A_{i,j}$

$$\rightsquigarrow c(i, j) = \begin{cases} 0, & \text{if } A_{i,j} = \emptyset; \\ \max\{c(i, k) + c(k, j) + 1 : a_k \in A_{i,j}\} & \text{otherwise.} \end{cases}$$

4.–6. *Omitted* (could be done following the standard scheme)

- Problem-specific insight from Unit 11 \rightsquigarrow Can always use $k = \min\{k : a_k \in A_{ij}\}$
(earliest finish time)

No guess needed!

12.4 The Bellman-Ford Algorithm

Recall Shortest Paths

► Single Source Shortest Path Problem (SSSPP)

► **Given:** directed, edge-weighted, simple graph $G = (V, E, c)$
with edge costs $c : E \rightarrow \mathbb{R}$, a start vertex $s \in V$

► **Goal:** a data structure that reports for every $v \in V$:
 $\delta_G(s, v)$: the shortest-path distance from s to v
spath(v): a shortest path from s to v (if it exists)

► $\delta_G(s, v) = \boxed{\inf (\{+\infty\} \cup \{c(w) : w \text{ an } s\text{-}v\text{-walk in } G\})}$

► Write δ instead of δ_G when graph clear from context

► Here: Assume **negative-weight edges** are present (otherwise Dijkstra suffices)

► but for now: assume there is **no negative cycle**

$\rightsquigarrow \delta(s, v) > -\infty$ and can restrict to shortest **paths** (not walks)

Shortest Paths as DP – Last Edge Decomposition

- ▶ Idea: Every nontrivial shortest path has a **last edge**. *We don't know which; so guess!*

↪ Subproblems: for $w \in V$, compute $\delta(s, w)$.

↪ Recurrence: $\delta(s, w) = \min\{\delta(s, v) + c(vw) : vw \in E\}$



subproblem dependency graph is isomorphic to G^T ! ↪ doesn't work in general

↪ Yields usable (terminating!) algorithm iff G is a DAG.



To break the cycles, let's turn them into a helix!

- ▶ Need to build “layers” in the subproblem dependency graph, so that edges can't go back up.
- ▶ **Subproblems:** (w, ℓ) for $w \in V$, $\ell \in [0..n)$, compute $\delta_{\leq \ell}(s, w)$
where $\delta_{\leq \ell}(s, v) = \min(\{+\infty\} \cup \{c(w) : w \text{ an } s\text{-}v\text{-walk with } \leq \ell \text{ edges}\})$
- ▶ **Original problems:** $\ell = n - 1$ (without negative cycles, paths suffice)
- ▶ **Recurrence:**
$$\delta_{\leq \ell}(s, w) = \begin{cases} \infty & \text{if } \ell = 0 \text{ and } s \neq w \\ 0 & \text{if } \ell = 0 \text{ and } s = w \\ \min\{\delta_{\leq \ell-1}(s, v) + c(vw) : vw \in E\} & \text{otherwise} \end{cases}$$

Shortest Paths as DP – Length Layers

Hold On – What about negative cycles?

- ▶ The recurrence for $\delta_{\leq \ell}$ seems to work fine with *negative* edges
But G could contain a **negative-weight cycle** $C \dots$

$$\delta_{\leq \ell}(s, w) = \begin{cases} \infty & \text{if } \ell = 0 \text{ and } s \neq w \\ 0 & \text{if } \ell = 0 \text{ and } s = w \\ \min\{\delta_{\leq \ell-1}(s, v) + c(vw) : vw \in E\} & \text{otherwise} \end{cases}$$



Isn't that a contradiction to the non-existence of shortest paths?

- ▶ No. If we restrict the length, shortest walks always exist.
- ▶ But: If there is a negative cycle $C[0..k]$ with paths $s \rightsquigarrow C$ and $C \rightsquigarrow w$,
then $\delta_{\leq \ell}(s, w) > \delta_{\leq \ell+k}(s, w) > \delta_{\leq \ell+2k}(s, w) > \dots$ (and $\delta(s, w) = -\infty$)
- \rightsquigarrow We can *detect* if any negative cycle is reachable from s by including more layers $\ell \geq n$ and check if some vertex still improves.
 - ▶ *How many further layers do we need / when is it safe to stop?*

Detecting negative cycles

We can detect reachable negative cycles by including just the *single* extra layer $\ell = n!$

Lemma: $\exists w : \delta_{\leq n}(s, w) < \delta_{\leq n-1}(s, w)$ iff negative cycle reachable from s

- “ \Rightarrow ”
- ▶ If some vertex w improves further, i. e., $\delta_{\leq n}(s, w) < \delta_{\leq n-1}(s, w)$
a walk $W[0..n]$ with $c(W) = \delta_{\leq n}(s, w)$ was the **shortest** way to reach w
 - \rightsquigarrow W is a non-simple walk, i. e., it contains a cycle
 - ▶ Let $P[0..k]$ be the path resulting from W by shortcutting all cycles $\rightsquigarrow k \leq n - 1$
 - $\rightsquigarrow c(P) \geq \delta_{\leq n-1}(s, w) > \delta_{\leq n}(s, w) = c(W)$
 - $\rightsquigarrow \exists$ negative cycle reachable from s
- “ \Leftarrow ”
- ▶ Conversely, let negative cycle $C[0..k]$ be reachable from s
 - $\rightsquigarrow c(C) = \sum_{i=0}^{k-1} c(C[i]C[i+1]) < 0$
 - ▶ Assume towards a contradiction that $\forall w : \delta_{\leq n}(s, w) = \delta_{\leq n-1}(s, w)$
 - $\rightsquigarrow \forall vw \in E : \delta_{\leq n-1}(s, w) \leq \delta_{\leq n-1}(s, v) + c(vw)$ (no update in layer $\ell = n$)
 - ▶ summing this inequality over $C[0..k]$ yields (abbreviating $\delta(w) := \delta_{\leq n-1}(s, w)$)
- $$\sum_{i=1}^k \delta(C[i]) \leq \sum_{i=1}^k \left(\delta(C[i-1]) + c(C[i]C[i+1]) \right) = \sum_{i=0}^{k-1} \delta(C[i]) + \underbrace{\sum_{i=1}^k c(C[i]C[i+1])}_{= c(C)} < 0$$
- $\rightsquigarrow 0 \leq c(C) < 0$ ⚡

Shortest Paths as DP – Template Algorithm

► Strictly following the template works ...

- Subproblem order: by increasing $\ell \in [0..n]$ and $v \in V$
- Bottom-up table filling:

```
1 procedure shortestPathsDP( $G, s$ ):
```

```
2   // Base case  $\ell = 0$ :
```

```
3    $\delta[0..n][0..n] := +\infty$  //  $\delta[\ell][v]$  will store  $\delta_{\leq \ell}(s, v)$ 
```

```
4    $\delta[0][s] := 0$ 
```

```
5   for  $\ell := 1, \dots, n$  // layer
```

```
6     for  $w := 0, \dots, n - 1$  // vertex
```

```
7       for  $vw \in E$ 
```

```
8          $\delta[\ell][w] := \min\{\delta[\ell][w], \delta[\ell - 1][v] + c(vw)\}$ 
```

```
9   return  $\delta$ 
```

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

$$\delta_{\leq \ell}(s, w) = \begin{cases} \infty & \text{if } \ell = 0 \text{ and } s \neq w \\ 0 & \text{if } \ell = 0 \text{ and } s = w \\ \min\{\delta_{\leq \ell-1}(s, v) + c(vw) : vw \in E\} & \text{otherwise} \end{cases}$$

► ... but some improvements are possible!

- Iterating over *incoming* edges is not convenient

↪ order of updates within layer ℓ doesn't matter ↪ iterate forwards!

- only use final distances in the end; we waste space by keeping 2D array around

↪ can actually just do updates in place, using a single array δ

↪ Don't strictly solve subproblems (ℓ, v) any more! (but final result correct)

The Bellman-Ford Algorithm

```
1 procedure bellmanFord( $G, s$ ):
2    $dist[0..n] := +\infty$ ;  $pred[0..n] := null$ 
3    $dist[s] := 0$ 
4   for  $\ell := 1, \dots, n - 1$ 
5     for  $v := 0, \dots, n - 1$ 
6       for  $(w, c) \in G.adj[v]$ 
7         if  $dist[w] > dist[v] + c$ 
8            $dist[w] := dist[v] + c$ 
9            $pred[w] := v$  // remember for backtrace
10  for  $v := 0, \dots, n - 1$ 
11    for  $(w, c) \in G.adj[v]$ 
12      if  $dist[w] > dist[v] + c$ 
13        return HAS_NEGATIVE_CYCLE
14  return ( $dist, pred$ )
```

- ▶ Final algorithm
(including shortest path tree via *pred*)
- ▶ **Correctness:**
 - ▶ by induction over loop iteration show $dist[w] \leq \delta_{\leq \ell}(s, w)$ and if finite, $dist[w]$ is $c(P)$ for some s - w -path
 - ▶ negative cycle detection from Lemma
- ▶ **Space:** $\Theta(n)$
- ▶ **Running time:** $O(n(n + m))$

Extensions:

- ▶ Can be implemented in $O(nm)$ time by removing unreachable vertices from consideration
- ▶ Instead of only detecting a negative cycle, we can return one;
we can also explicitly find all vertices with $\delta(s, w) = -\infty$ (needs another traversal).
- ▶ Can terminate with smaller ℓ if no distance changed \rightsquigarrow faster for some graphs

12.5 Making Change in Pre-1971 UK

Recall Unit 11

Greed For Change

The Change-Making Problem (a.k.a. Coin-Exchange Problem)

- ▶ **Given:** a set of integer denominations of coins $w_1 < w_2 < \dots < w_k$ with $w_1 = 1$, target value $n \in \mathbb{N}_{\geq 1}$ (we have sufficient supply of all coins ...)
- ▶ **Goal:** “fewest coins to give change n ”, i.e., multiplicities $c_1, \dots, c_k \in \mathbb{N}_{\geq 0}$ with $\sum_{i=1}^k c_i \cdot w_i = n$ minimizing $\sum_{i=1}^k c_i$

For Euro coins, denominations are $\textcircled{1\text{€}}$, $\textcircled{2\text{€}}$, $\textcircled{5\text{€}}$, $\textcircled{10\text{€}}$, $\textcircled{20\text{€}}$, $\textcircled{50\text{€}}$, $\textcircled{1\text{€}}$, and $\textcircled{2\text{€}}$.
formally: $1, 2, 5, 10, 20, 50, 100$, and 200 .
 $w_1 \quad w_2 \quad w_3 \quad w_4 \quad w_5 \quad w_6 \quad w_7 \quad w_8$

~> Simple greedy algorithm:
largest coins first

- ▶ optimal time ($O(k)$ if coins sorted)
- ▶ is $\sum c_i$ minimal?

```
1 procedure greedyChange( $w[1..k], n$ ):  
2   // Assumes  $1 = w[1] < w[2] < \dots < w[k]$   
3   for  $i := k, k-1, \dots, 1$ :  
4      $c[i] := \lfloor n/w[i] \rfloor$   
5      $n := n - c[i] \cdot w[i]$   
6   // Now  $n == 0$   
7   return  $c[1..k]$ 
```

Pre-Decimal English Coins

*We discussed that for some (unwise) choices of denominations, Greedy cannot give optimal change.
Welcome to Britain until 1971!*

British Pre-Decimal Coins:

- ▶ $\frac{1}{2}$ penny,
- ▶ 1 penny,
- ▶ 3 pence,
- ▶ 6 pence,
- ▶ shilling = 12 pence,
- ▶ florin = 24 pence
- ▶ half-crown = 30 pence
- ▶ crown = 60 pence
- ▶ pound = 240 pence
- ▶ guinea = $21 \cdot 12 = 252$ pence
(obsolete as coin since 1816)

↪ Greedy would give 48 pence
as 30p + 12p + 6p

▶ obviously, 2 florins are more efficient

↪ How to solve exactly?

As the old saying goes . . .

Where Greedy fails, DP prevails.

(but mind details, and how it scales)

Making Change by DP

Idea: Every solution must pick a first coin. Which one? Unclear, so guess!

- ▶ **Subproblems:** Change for $m \in [0..n]$ (with coins w_1, \dots, w_k)
Original problem $m = n$
- ▶ **Guess:** first coin w_i to use
- ▶ **Recurrence** $C(m)$ = smallest #coins to give change m

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

$$C(m) = \begin{cases} 0 & \text{if } m = 0 \\ 1 + \min\{C(m - w_i) : i \in [1..k] \wedge w_i \leq m\} & \text{otherwise} \end{cases}$$

▶ Bottom-up implementation & Backtrace

```
1 procedure dpChange( $w[1..k], n$ ):
2    $C[0..n] := +\infty$ 
3    $C[0] := 0$ 
4   for  $m := 1, \dots, n$ 
5     for  $i := 1, \dots, k$ 
6       if  $w[i] \geq m$ 
7          $q := 1 + C[m - w[i]]$ 
8          $C[m] := \min\{C[m], q\}$ 
9   return  $C[n]$ 
```

```
1 procedure tracebackChange( $w[1..k], n$ ):
2    $C[0..n] := \text{dpChange}(w[1..k], n)$ 
3    $c[1..k] := 0$  // coin multiplicities
4    $m := n$ 
5   while  $m > 0$ 
6     for  $i := 1, \dots, k$ 
7       if  $w[i] \geq m \wedge C[m] == 1 + C[m - w[i]]$ 
8          $c[i] := c[i] + 1$ ;  $m := m - w[i]$ 
9   return  $c[1..k]$ 
```

Making Change by DP – Analysis

- **Input:** denominations of coins
 $w_1 < w_2 < \dots < w_k$ with $w_1 = 1$,
 target value $n \in \mathbb{N}_{\geq 1}$

- ▶ **Space:** $\Theta(n)$
- ▶ **Running Time:** $O(n \cdot k)$
 - #subproblems (points to n)
 - time per subproblem (points to k)

```

1 procedure dpChange( $w[1..k], n$ ):
2    $C[0..n] := +\infty$ 
3    $C[0] := 0$ 
4   for  $m := 1, \dots, n$ 
5     for  $i := 1, \dots, k$ 
6       if  $w[i] \geq m$ 
7          $q := 1 + C[m - w[i]]$ 
8          $C[m] := \min\{C[m], q\}$ 
9   return  $C[n]$ 

```

How good is this running time?

- ▶ A linear function in both input numbers seems decent, right? (If k and n small, certainly Yes.)
 - ▶ Running time is also certainly a *polynomial* in n and k
- ▶ But: In terms of *computational complexity*, dpChange is an **exponential-time algorithm**!
 - ▶ Reason: We give the input **number** n in **binary**, so n is exponential in its *input size*.

⚠ Must distinguish: *value* of a number (in the input) vs. *size* of the (encoding of the) input

dpChange is a *pseudo-polynomial time* algorithm

- ▶ Actually, the general making-change problem is NP-complete (!)

Knapsack

Let's look at slightly more interesting problem: *Knapsack* („*Rucksack*“).

The 0/1-Knapsack Problem

a.k.a. the burglar's problem

- ▶ **Given:** k items with weights $w_1, \dots, w_k \in \mathbb{N}_{\geq 1}$ and values $v_1, \dots, v_k \in \mathbb{R}_{\geq 0}$; a weight budget $W \in \mathbb{N}$
- ▶ **Goal:** Subset $I \subseteq [1..k]$ such that $\sum_{i \in I} w_i \leq W$ with maximum $\sum_{i \in I} v_i$.

Variant closer to Making change: Can use each item several times

- ▶ Recall from tutorials: Greedy fails miserably in general.

⇒ Let's try DP!

- ▶ **Subproblems:** $B \in [0..W]$, best value with total weight $\leq B$
- ▶ **Guess:** first item i with $w_i \leq B$.

⚡ Subproblem not of same type since w_i no longer there!

⇒ 2^k possible “states” to be in (items already used) (**0/1**-Knapsack)

⚡ need a table of size $W \cdot 2^k \dots$ might as well do brute force then!

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

Knapsack by DP

↪ Force order to consider items in!

► Let's refine the guessing part to

Guess: Whether or not to include the *last* item (k)

↪ For subproblem, restrict to items $1, \dots, k-1$ (in either case)

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace

↪ **Subproblems:** (ℓ, B) for $\ell \in [1..k]$ and $B \in [0..W]$

$$V(\ell, B) = \max_I \sum_{i \in I} v_i \text{ over sets of items } I \subset [1..\ell] \text{ with } \sum_{i \in I} w_i \leq B$$

Original problem corresponds to $V(k, W)$

$$\text{► Recurrence: } V(\ell, B) = \begin{cases} 0 & \text{if } \ell = 1 \wedge w_1 > B \\ v_1 & \text{if } \ell = 1 \wedge w_1 \leq B \\ \max \left\{ \overset{\text{take item } \ell}{v_\ell + V(\ell-1, B-w_\ell)}, \overset{\text{don't take } \ell}{V(\ell-1, B)} \right\} & \text{otherwise} \end{cases}$$



Cookie-Cutter Steps 4. – 6. Omitted

► $V(\ell, \cdot)$ only needs $V(\ell-1, \cdot)$ ↪ two arrays $V[0..W]$ and $V_{\text{prev}}[0..W]$ suffice

↪ $\Theta(W)$ space, $\Theta(W \cdot k)$ time (pseudo-polynomial algorithm)

12.6 Optimal Merge Trees & Optimal BSTs

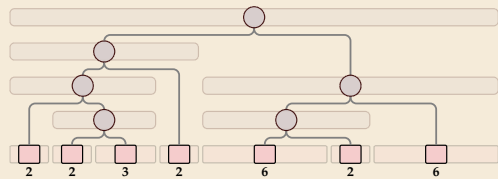
Recall Unit 4


Good merge orders


◀◀ Let's take a step back and breathe.

► Conceptually, there are two tasks:

1. Detect and use existing runs in the input $\rightsquigarrow \ell_1, \dots, \ell_r$ (easy) ✓
2. Determine a favorable **order of merges of runs** ("automatic" in top-down mergesort)



Merge cost = total area of 
= total length of paths to all array entries
= $\sum_{w \text{ leaf}} \text{weight}(w) \cdot \text{depth}(w)$

\rightsquigarrow *optimal* merge tree 
= optimal *binary search tree*
for leaf weights ℓ_1, \dots, ℓ_r
(optimal expected search cost)

well-understood problem
with known algorithms

Optimal Alphabetic Trees

“well-understood problem with known algorithms” ... let's make it so 😊

- ▶ **Given:** Leaf weights ℓ_0, \dots, ℓ_n normalized to $\ell_0 + \dots + \ell_n = 1$
- ▶ **Goal:** Binary search tree T with $n + 1$ null pointers L_0, \dots, L_n , such that

$$c(T) := \sum_{i=1}^n \ell_i \cdot \text{depth}_T(L_i) \text{ is minimized}$$

▶ Equivalent interpretations:

1. *Optimal Static BST* with keys $1, 2, \dots, n$

\rightsquigarrow leaf L_i reached when searching for $i + 0.5$ \rightsquigarrow $c(T)$ *expected cost of unsuccessful search*

#comparisons
↓

2. *Alphabetic code* for $\sigma = n + 1$ symbols; like Huffman code, but *codewords must retain order* (if $i < j$ then the codeword for i lexicographically smaller than codeword for j)

\rightsquigarrow $c(T)$ *expected codeword length*

▶ Inherit lower bound from Huffman codes: $c(T) \geq \mathcal{H}$ with $\mathcal{H} = \sum_{i=0}^n \ell_i \cdot \log_2 \left(\frac{1}{\ell_i} \right)$

3. *Merge tree* for adaptive sorting; $c(T) =$ merge cost per element.

- ▶ Via Peeksor or Powersort know methods to achieve $c(T) \leq \mathcal{H} + 2$
- ▶ But neither are in general optimal

Optimal Alphabetic Trees by DP

► **Guess:** (Key in) root $r \in [1..n]$ of BST T (= #leaves in left subtree)

► **Subproblems:** $[i..j]$ for $0 \leq i < j \leq n + 1$

$C(i, j)$ = cost of opt. BST with leaf weights $\ell_i, \dots, \ell_{j-1}$

Original problem: $C(0, n + 1)$

► **Recurrence:**

$$C(i, j) = \begin{cases} 0 & \text{if } j - i = 1 \\ \ell_i + \dots + \ell_{j-1} + \min\{C(i, r) + C(r, j) : r \in [i + 1..j - 1]\} & \text{otherwise} \end{cases}$$

all leaves in subtree pay 1 at root. . .

... plus cost to continue in left/right subtree

1. Subproblems
2. Guess!
3. DP Recurrence
4. Memoization
5. Table Filling
6. Backtrace



~> Obtain a $O(n^3)$ time and $O(n^2)$ space algorithm

Optimal Binary Search Trees

- ▶ Algorithm can be generalized to Optimal BSTs when also internal nodes have weights
 - ▶ Same DP subproblems
- ▶ Running time can be reduced to $O(n^2)$ using *quadrangle inequality*
 - ▶ Intuitively: When adding more weight in right subtree, optimal root cannot move left.
 - ▶ Requires to remember r for each subproblem
- ▶ For original alphabetic tree problem, can actually find optimal tree in $O(n \log n)$ time with a much more intricate algorithm

12.7 Edit Distance

Edit Distance

Our last DP application here: (algorithmic foundation of) `diff`!

- ▶ `diff` is a classic Unix tool to compare two text files
- ▶ routinely used in version control systems such as `git`
- ▶ abstract problem: measure how different two strings are
 - ▶ We've seen *Hamming distance* . . .
But how to deal with strings of different lengths?
 - ▶ how to match common parts that are far apart?
 - ▶ `diff` works line-oriented, but we will formulate the problem character oriented

Edit Distance Problem

- ▶ **Given:** String $A[0..m)$ and $B[0..n)$ over alphabet $\Sigma = [0..\sigma)$.
- ▶ **Goal:** $d_{\text{edit}}(A, B) =$ minimal # symbol operations to transform A into B
operations can be insertion/deletion/substitution of single character

Edit Distance Example

Example: edit distance $d_{\text{edit}}(\text{algorithm}, \text{logarithm})$?

algorithm

logarithm

0123456789

a l · g o r i t h m

- | + | X | | | | |

· l o g a r i t h m

Edit Distance by DP

1. **Subproblems:** (i, j) for $0 \leq i \leq m, 0 \leq j \leq m$ compute $d_{\text{edit}}(A[0..i], B[0..j])$
2. **Guess:** What to do with last positions? (insert/delete/(mis)match)
3. **Recurrence:** $D(i, j) = d_{\text{edit}}(A[0..i], B[0..j])$

$$D(i, j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} D(i-1, j) + 1, \\ D(i, j-1) + 1, \\ D(i-1, j-1) + [A[i-1] \neq B[j-1]] \end{cases} & \text{otherwise} \end{cases}$$

$\rightsquigarrow O(nm)$ space and time

space can be improved to $O(\min\{n, m\})$ by remembering only 2 rows or columns

4. An optimal *edit script* can be constructed by a backtrace

Generalized Edit Distances

- ▶ The variant we discussed is also called *Levenshtein distance*
 - ▶ all operation have cost 1
- ▶ we can directly give each of the following its **own cost** in our DP algorithm
 - ▶ deleting an occurrence of $a \in \Sigma$
 - ▶ inserting an $a \in \Sigma$
 - ▶ substituting $a \in \Sigma$ for $b \in \Sigma$
- ▶ Extensions of the algorithm can support:
 - ▶ **free** insert/delete at beginning/end of a string
 - ▶ *affine gap costs*, i. e., inserting/deleting k **consecutive** chars costs $c \cdot k + d$ for constants c and d
- ▶ extensions widely used to find approximate matches, e. g., in DNA sequences

Dynamic Programming – Summary

- 👍 Versatile and powerful algorithm design paradigm
- 👍 Once key idea (recurrence) clear, implementation rather straight-forward

