paradigms:
- recursion
- dynamic programming

problems:
- fibonacci numbers
- edit distance
- knapsack
Recursion

Definition

A **reduction** transforms a given problem into a yet another problem, possibly into **several instances** of another problem. **Recursion** is a reduction from one instance of a problem to instances of the **same** problem.

**example** *(Karatsuba, Strassen, ...):*

- reduce problem instances of size $n$ to problem instances of size $n/2$
- terminate recursion at $O(1)$-size problem instances, solve straightforwardly as a *base case*
recursive paradigms:

- **tail recursion**: expend effort to reduce given problem to *single* (smaller) problem. Often can be reformulated as a non-recursive algorithm (iterative, or greedy).

- **divide and conquer**: expend effort to reduce (divide) given problem to *multiple, independent* smaller problems, which are solved separately. Solutions to smaller problems are combined to solve original problem (conquer). For example: Karatsuba, Strassen, …

- **dynamic programming**: expend effort to reduce given problem to multiple *correlated* smaller problems. Naive recursion often *not* efficient, use **memoization** to avoid wasteful recomputation.
Recursion (II)

\[
\text{foo}\text{\(instance\; X\)} \\
\text{if } X \text{ is a base case then} \\
\quad \text{solve it and return solution} \\
\text{else} \\
\quad \text{do stuff} \\
\quad \text{foo}\text{\(X_1\)} \\
\quad \text{do stuff} \\
\quad \text{foo}\text{\(X_2\)} \\
\quad \text{foo}\text{\(X_3\)} \\
\quad \text{more stuff} \\
\quad \text{return solution for } X
\]

analysis:
- \textit{recursion tree}: each instance } X \text{ spawns \textit{new} children } X_1, X_2, X_3
- \textit{dependency graph}: each instance } X \text{ links to sub-problems } X_1, X_2, X_3
The Fibonacci sequence $F_0, F_1, F_2, F_3, \ldots \in \mathbb{N}$ is the sequence of numbers defined by

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$

**Remarks:**

- Arises in surprisingly many places — the journal *The Fibonacci Quarterly*
- $F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, $\varphi$ is the *golden ratio* $\varphi := \frac{1+\sqrt{5}}{2} \approx 1.618 \cdots$
- $1 - \varphi \approx -0.618 \cdots \Rightarrow |(1 - \varphi)^n| \leq 1$, and further $(1 - \varphi)^n \to_{n\to\infty} 0$
- $F_n = \Theta(\varphi^n)$. 

$\varphi$ is the golden ratio.
**Fibonacci Numbers (II)**

**question:** given \( n \), compute \( F_n \).

**answer:**

```python
fib(n):
    if (n = 0)
        return 0
    else-if(n = 1)
        return 1
    else
        return fib(n-1) + fib(n-2)
```

**correctness:** clear

**complexity:** let \( T(n) \) denote the number of additions. Then

- \( T(0) = 0, \ T(1) = 0 \)
- \( T(2) = 1, \)
- \( T(n) = T(n-1) + T(n-2) \)
- \( \Rightarrow \ T(n) = F_{n-1} = \Theta(\varphi^n) \Rightarrow \text{exponential time!} \)
Fibonacci Numbers (III)

**recursion tree:** for $F_4$

![Recursion Tree](image)

**dependency graph:** for $F_4$

![Dependency Graph](image)
Fibonacci Numbers (IV)

iterative algorithm:

```python
fib-iter(n):
    if n = 0
        return 0
    if n = 1
        return 1
    F[0] = 0
    F[1] = 1
    for 2 ≤ i ≤ n
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

correctness: clear

complexity: $O(n)$ additions

remarks:
- $F_n = \Theta(\phi^n) \implies F_n$ takes $\Theta(n)$ bits $\implies$ each addition takes $\Theta(n)$ steps
- $\implies O(n^2)$ is the actual runtime
recursive paradigms for $F_n$:

- **naive recursion**: recurse on subproblems, solves the *same* subproblem multiple times
- **iterative algorithm**: stores solutions to subproblems to avoid recomputation — **memoization**

**Definition**

*Dynamic programming* is the method of speeding up naive recursion through memoization.

**remarks:**

- If number of subproblems is polynomially bounded, often implies a polynomial-time algorithm
- Memoizing a recursive algorithm is done by tracing through the dependency graph
question: how to memoize exactly?

```python
def fib(n):
    if n == 0
        return 0
    if n == 1
        return 1
    if fib(n) was previously computed
        return stored value fib(n)
    else
        return fib(n-1) + fib(n-2)
```

question: how to memoize exactly?

- explicitly: just do it!
- implicitly: allow clever data structures to do this automatically
global F[.]
fib(n):
  if $n = 0$
    return 0
  if $n = 1$
    return 1
  if $F[n]$ initialized
    return $F[n]$
  else
    $F[n] = \text{fib}(n - 1) + \text{fib}(n - 2)$
    return $F[n]$

- **explicit** memoization: we decide *ahead* of time what types of objects $F$ stores
  - e.g., $F$ is an array
  - requires more deliberation on problem structure, but can be more efficient
- **implicit** memoization: we let the data structure for $F$ handle whatever comes its way
  - e.g., $F$ is an dictionary
  - requires *less* deliberation on problem structure, and can be less efficient
  - sometimes can be done automatically by functional programming languages (LISP, etc.)
**question:** how much *space* do we need to memoize?

```python
fib-iter(n):
    if n = 0
        return 0
    if n = 1
        return 1
    F_prev = 1
    F_prevprev = 0
    for 2 ≤ i ≤ n
        F_cur = F_prev + F_prevprev
        F_prevprev = F_prev
        F_prev = F_cur
    return F_cur
```

correctness: clear

**complexity:** $O(n)$ additions, $O(1)$ numbers stored
Memoization (IV)

Definition

**Dynamic programming** is the method of speeding up naive recursion through memoization.

goals:

- Given a recursive algorithm, analyze the complexity of its memoized version.
- Find the *right* recursion that can be memoized.
- Recognize when dynamic programming will efficiently solve a problem.
- Further optimize time- and space-complexity of dynamic programming algorithms.
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. The edit distance between $x$ and $y$ is the minimum number of insertions, deletions and substitutions required to transform $x$ into $y$.

Example

money  boney  bone  bona  boa  boba $\implies$ edit distance $\leq 5$

remarks:

- edit distance $\leq 4$
- intermediate strings can be arbitrary in $\Sigma^*$
Definition

Let $x, y \in \Sigma^*$ be two strings over the alphabet $\Sigma$. An **alignment** is a sequence $M$ of pairs of indices $(i, j)$ such that

- an index could be empty, such as $(, 4)$ or $(5, )$
- each index appears exactly once per coordinate
- no crossings: for $(i, j), (i', j') \in M$ either $i < i'$ and $j < j'$, or $i > i'$ and $j > j'$

The **cost** of an alignment is the number of pairs $(i, j)$ where $x_i \neq y_j$.

Example

```
mon ey
bo ba
```

$M = \{(1, 1), (2, 2), (3, ), (3, ), (4, 4), (5, )\}$, cost 5
question: given two strings $x, y \in \Sigma^*$, compute their edit distance

Lemma

The edit distance between two strings $x, y \in \Sigma^*$ is the minimum cost of an alignment.

Proof.

Exercise.

question: given two strings $x, y \in \Sigma^*$, compute the minimum cost of an alignment

remarks:

- can also ask to compute the alignment itself
- widely solved in practice, e.g., the BLAST heuristic for DNA edit distance
Lemma

Let $x, y \in \Sigma^*$ be strings, and $a, b \in \Sigma$ be symbols. Then

$$\text{dist}(x \circ a, y \circ b) = \min \left\{ \begin{array}{ll}
\text{dist}(x, y) + 1 & [a \neq b] \\
\text{dist}(x, y \circ b) + 1 \\
\text{dist}(x \circ a, y) + 1 
\end{array} \right. $$

Proof.

In an optimal alignment from $x \circ a$ to $y \circ b$, either:

- $a$ aligns to $b$, with cost $1[a \neq b]$
- $a$ is deleted, with cost 1
- $b$ is deleted, with cost 1
recursive algorithm:

\[
\text{dist}(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_n)
\]

\[
\text{if } n = 0 \text{ return } m
\]

\[
\text{if } m = 0 \text{ return } n
\]

\[
d_1 = \text{dist}(x_{<n}, y_{<m}) + 1[ x_n \neq y_m ]
\]

\[
d_2 = \text{dist}(x_{<n}, y) + 1
\]

\[
d_3 = \text{dist}(x, y_{<m}) + 1
\]

\[
\text{return min}(d_1, d_2, d_3)
\]

correctness: clear

complexity: ???
Edit Distance (VI)

(ab, bab) is repeated!

**memoization:** define subproblem \((i, j)\) as computing \(\text{dist}(x_{\leq i}, y_{\leq j})\)
memoized algorithm:

```
global d[·][·]
dist(x_1x_2 \cdots x_n, y_1y_2 \cdots y_m, (i,j))
    if d[i][j] initialized
        return d[i][j]
    if i = 0
        d[i][j] = j
    else-if j = 0
        d[i][j] = i
    else
        d_1 = dist(x,y,(i-1,j-1)) + 1[x_i \neq y_j]
        d_2 = dist(x,y,(i-1,j)) + 1
        d_3 = dist(x,y,(i,j-1)) + 1
        d[i][j] = min(d_1, d_2, d_3)
    return d[i][j]
```
dependency graph:
Iterative algorithm:

\[
\text{dist}(x_1x_2\cdots x_n, y_1y_2\cdots y_m)
\]
for \(0 \leq i \leq n\)
\[
d[i][0] = i
\]
for \(0 \leq j \leq m\)
\[
d[0][j] = j
\]
for \(0 \leq i \leq n\)
\[
\text{for } 0 \leq j \leq m
\]
\[
d[i][j] = \min\left\{d[i - 1][j - 1] + 1, d[i - 1][j] + 1, d[i][j - 1] + 1\right\}
\]

Correctness: clear

Complexity: \(O(nm)\) time, \(O(nm)\) space
Corollary

Given two strings \( x, y \in \Sigma^* \) can compute the minimum cost alignment in \( O(nm) \)-time and -space.

Proof.

Exercise. *Hint*: follow how each subproblem was solved.
Dynamic Programming

template:
- develop recursive algorithm
- understand structure of subproblems
- memoize
  - implicitly, via data structure
  - explicitly, converting to iterative algorithm to traverse dependency graph via topological sort
- analysis (time, space)
- further optimization
the knapsack problem:

**input:** knapsack capacity $W \in \mathbb{N}$ (in pounds). $n$ items with weights $w_1, \ldots, w_n \in \mathbb{N}$, and values $v_1, \ldots, v_n \in \mathbb{N}$.

**goal:** a subset $S \subset [n]$ of items that fit in the knapsack, with maximum value

$$\max_{S \subseteq [n]} \sum_{i \in S} v_i \quad \text{subject to} \quad \sum_{i \in S} w_i \leq W$$

**remarks:**

- prototypical problem in *combinatorial optimization*, can be generalized in numerous ways
- needs to be solved in practice
Example

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>value</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
</tbody>
</table>

For $W = 11$, the best is $\{3, 4\}$ giving value 40.

Definition

In the special case of when $v_i = w_i$ for all $i$, the knapsack problem is called the **subset sum** problem.
Knapsack (III)

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>6</td>
<td>16</td>
<td>22</td>
<td>28</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

and weight limit $W = 15$. What is the best solution value?

(a) 22
(b) 28
(c) 38
(d) 50
(e) 56
**Knapsack (IV)**

**greedy approaches:**

- greedily select by maximum value:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For \( W = 2 \), greedy-value will pick \{3\}, but optimal is \{1, 2\}

- greedily select by minimum weight:
  
<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>weight</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

  For \( W = 2 \), greedy-weight will pick \{1\}, but optimal is \{2\}

- greedily select by maximum value/weight ratio:

<table>
<thead>
<tr>
<th>item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>weight</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

  For \( W = 4 \), greedy-value will pick \{3\}, but optimal is \{1, 2\}

**remark:** while greedy algorithms fail to get the *best* result, they can still be useful for getting solutions that are *approximately* the best
Lemma

Consider the instance $W$, $(v_i)_{i=1}^n$, and $(w_i)_{i=1}^n$, with optimal solution $S \subseteq [n]$. Then,

1. if $n \not\in S$, then $S \subseteq [n - 1]$ is an optimal solution for the knapsack instance $(W, (v_i)_{i<n}, (w_i)_{i<n})$.
2. if $n \in S$, then $S \setminus \{n\} \subseteq [n - 1]$ is an optimal solution for the knapsack instance $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$.

Proof.

1. Any $S \subseteq [n - 1]$ feasible for $(W, (v_i)_{i<n}, (w_i)_{i<n})$, will also satisfy the original weight constraint.
2. Any $S \subseteq [n - 1]$ feasible for $(W - w_n, (v_i)_{i<n}, (w_i)_{i<n})$, will have that $S \cup \{n\}$ will also satisfy the original weight constraint.
Fix an instance $W, v_1, \ldots, v_n, \text{ and } w_1, \ldots, w_n$. Define $\text{OPT}(i, w)$ to be the maximum value of the knapsack instance $w, v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then, 

$$\text{OPT}(i, w) = \begin{cases} 0 & i = 0 \\ \text{OPT}(i - 1, w) & w_i > w \\ \max \begin{cases} \text{OPT}(i - 1, w) \\ \text{OPT}(i - 1, w - w_i) + v_i \end{cases} & \text{else} \end{cases}$$

$\implies$ from instance $W, v_1, \ldots, v_n, \text{ and } w_1, \ldots, w_n$ we generate $O(n \cdot W)$-many subproblems $(i, w)_{i \in [n], w \leq W}$. 
Knapsack (VII)

an iterative algorithm: $M[i, w]$ will compute $\text{OPT}(i, w)$

for $0 \leq w \leq W$
  $M[0, w] = 0$
for $1 \leq i \leq n$
  for $1 \leq w \leq W$
    if $w_i > w$
      $M[i, w] = M[i - 1, w]$
    else
      $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$

correctness: clear

complexity:
  - $O(nW)$ time, but input size is $O(n + \log W + \sum_{i=1}^{n} (\log v_i + \log w_i))$

- e.g., $W = 2^n$ has $O(n)$ bits but requires $\Omega(2^n)$ runtime $\implies$ running time is not polynomial in the input
- Algorithm is pseudo-polynomial: running time is polynomial in magnitude of the input numbers
- Knapsack is NP-hard in general $\implies$ no efficient algorithm is expected to compute the exact optimum

punchline: had to correctly parameterize knapsack sub-problems $(v_j)_{j \leq i}, (w_j)_{j \leq i}$ by also considering arbitrary $w$. This is a common theme in dynamic programming problems.
today:

- paradigms:
  - recursion
  - dynamic programming

- problems:
  - fibonacci numbers
  - edit distance
  - knapsack

next time: more dynamic programming