cs473: Algorithms

Lecture 3: Dynamic Programming

Michael A. Forbes Chandra Chekuri

University of Illinois at Urbana-Champaign

September 3, 2019

Today

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*

Recursion (II)

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)

```
foo(instance X)
    if X is a base case then
        solve it and return solution
    else
        do stuff
        foo(X_1)
        do stuff
        foo(X_2)
        foo(X_3)
        more stuff
        return solution for X
```

analysis:

- *recursion tree:* each instance X spawns *new* children X_1, X_2, X_3
- *dependency graph:* each instance X links to sub-problems X_1, X_2, X_3

Fibonacci Numbers

Definition (Fibonacci 1200, Pingala -200)

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*
- $F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, φ is the *golden ratio* $\varphi := \frac{1+\sqrt{5}}{2} \approx 1.618\cdots$
- $\implies 1 - \varphi \approx -.618\cdots \implies |(1-\varphi)^n| \leq 1$, and further $(1-\varphi)^n \to_{n\to\infty} 0$ $\implies F_n = \Theta(\varphi^n)$.

Fibonacci Numbers (II)

question: given n, compute F_n.

answer:

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

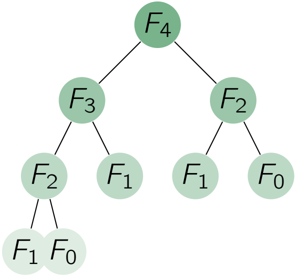
correctness: clear

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

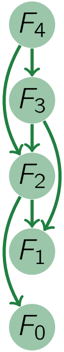
- $T(0) = 0,\ T(1) = 0$
- $T(2) = 1,$
- $T(n) = T(n-1) + T(n-2)$
- $\implies\ T(n) = F_{n-1} = \Theta(\varphi^n)\ \implies$ exponential time!

Fibonacci Numbers (III)

recursion tree: for F_4



dependency graph: for F_4



Fibonacci Numbers (IV)

iterative algorithm:

```
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:

- $F_n = \Theta(\varphi^n) \implies F_n$ takes $\Theta(n)$ bits $\implies$ each addition takes $\Theta(n)$ steps

$\implies$ $O(n^2)$ is the *actual* runtime

Memoization

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

Memoization (II)

question: how to memoize exactly?

```
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

Memoization (III)

```
global F[·]
fib(n):
    if n = 0
        return 0
    if n = 1
        return 1
    if F[n] initialized
        return F[n]
    else
        F[n] = fib(n − 1) + 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.)

Fibonacci Numbers (V)

question: how much *space* do we need to memoize?

fib-iter(n) :
  if $n = 0$
    return 0
  if $n = 1$
    return 1
  $F_{\texttt{prev}} = 1$
  $F_{\texttt{prevprev}} = 0$
  for $2 \leq i \leq n$
    $F_{\texttt{cur}} = F_{\texttt{prev}} + F_{\texttt{prevprev}}$
    $F_{\texttt{prevprev}} = F_{\texttt{prev}}$
    $F_{\texttt{prev}} = F_{\texttt{cur}}$
  return $F_{\texttt{cur}}$

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.

Edit Distance

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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 $\Longrightarrow$ edit distance ≤ 5

remarks:

- edit distance ≤ 4
- intermediate strings can be arbitrary in $\Sigma^\star$

Edit Distance (II)

Definition

Let $x, y \in \Sigma^\star$ be two strings over the alphabet Σ. 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

Edit Distance (III)

question: given two strings $x, y \in \Sigma^\star$, compute their edit distance

Lemma

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

Proof.

Exercise. $\square$

question: given two strings $x, y \in \Sigma^\star$, 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

Edit Distance (IV)

Lemma

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

$$\mathrm{dist}(x \circ a, y \circ b) = \min \begin{cases} \mathrm{dist}(x, y) + \mathbb{1}[\![a \neq b]\!] \\ \mathrm{dist}(x, y \circ b) + 1 \\ \mathrm{dist}(x \circ a, y) + 1 \end{cases}$$

Proof.

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

- a aligns to b, with cost $\mathbb{1}[\![a \neq b]\!]$
- a is deleted, with cost 1
- b is deleted, with cost 1 $\square$

Edit Distance (V)

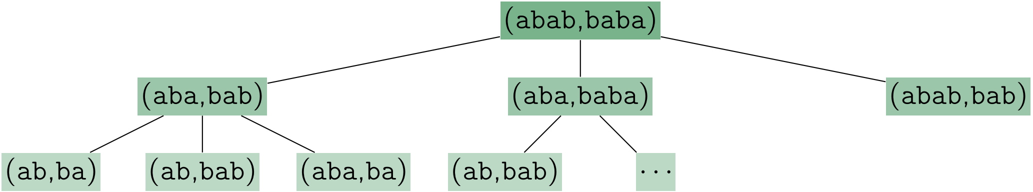
recursive algorithm:

$$
\begin{array}{l}
\textbf{dist}\,(x = x_1 x_2 \cdots x_n, y = y_1 y_2 \cdots y_n) \\
\quad \texttt{if}\ \ n = 0\ \texttt{return}\ \ m \\
\quad \texttt{if}\ \ m = 0\ \texttt{return}\ \ n \\
\quad d_1 = \textbf{dist}(x_{<n}, y_{<m}) + \mathbb{1}[\![x_n \neq y_m]\!] \\
\quad d_2 = \textbf{dist}(x_{<n}, y) + 1 \\
\quad d_3 = \textbf{dist}(x, y_{<m}) + 1 \\
\quad \texttt{return}\ \ \min(d_1, d_2, d_3)
\end{array}
$$

correctness: clear

complexity: ???

Edit Distance (VI)



(ab,bab) is *repeated*!

memoization: define subproblem (i, j) as computing $\mathrm{dist}(x_{\leq i}, y_{\leq y})$

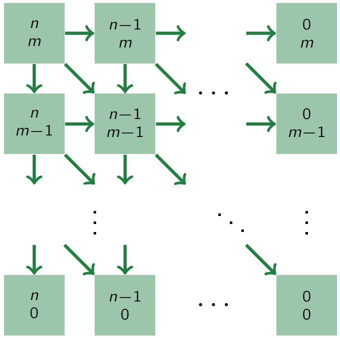
Edit Distance (VII)

memoized algorithm:

```
global d[·][·]
dist(x1x2···xn, y1y2···ym, (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
        d1 = dist(x, y, (i − 1, j − 1)) + 𝟙[[xi ≠ yj]]
        d2 = dist(x, y, (i − 1, j)) + 1
        d3 = dist(x, y, (i, j − 1)) + 1
        d[i][j] = min(d1, d2, d3)
    return d[i][j]
```

Edit Distance (VIII)

dependency graph:



Edit Distance (IX)

iterative algorithm:

dist $(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m)$

```
for 0 ≤ i ≤ n
    d[i][0] = i
for 0 ≤ j ≤ m
    d[0][j] = j
for 0 ≤ i ≤ n
    for 0 ≤ j ≤ m
```

$$d[i][j] = \min \begin{cases} d[i-1][j-1] + \mathbb{1}[\![x_i \neq y_j]\!] \\ d[i-1][j] + 1 \\ d[i][j-1] + 1 \end{cases}$$

correctness: clear

complexity: $O(nm)$ time, $O(nm)$ space

Edit Distance (X)

Corollary

Given two strings $x, y \in \Sigma^\star$ *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
  - implicity, via data structure
  - explicitly, converting to iterative algorithm to traverse dependency graph via topological sort
- analysis (time, space)
- further optimization

Knapsack

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

$$\max_{\substack{S \subseteq [n] \\ \sum_{i \in S} w_i \leq W}} \sum_{i \in S} v_i$$

remarks:

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

Knapsack (II)

Example

item	1	2	3	4	5
weight	1	2	5	6	7
value	1	6	18	22	28

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)

item	1	2	3	4	5
value	1	6	16	22	28
weight	1	2	5	6	7

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:

item	1	2	3
value	2	2	3
weight	1	1	2

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

- greedily select by minimum weight:

item	1	2
value	1	3
weight	1	2

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

- greedily select by maximum value/weight ratio:

item	1	2	3
value	3	3	5
weight	2	2	3

  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

Knapsack (V)

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* $n \notin S$, *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 □

Knapsack (VI)

Corollary

Fix an instance W, $v_1, \ldots, v_n$, and $w_1, \ldots, w_n$. Define $\mathrm{OPT}(i, w)$ to be the maximum value of the knapsack instance w, $v_1, \ldots, v_i$ and $w_1, \ldots, w_i$. Then,

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

$\implies$ from instance W, $v_1, \ldots, v_n$, 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 $\mathrm{OPT}(i, w)$

```
for 0 ≤ w ≤ W
    M[0, w] = 0
for 1 ≤ i ≤ n
    for 1 ≤ w ≤ 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 $(v_j)_{j \le i}$,$(w_j)_{j \le i}$ by *also* considering arbitrary w. This is a common theme in dynamic programming problems.

Today

today:

- paradigms:
  - recursion
  - dynamic programming
- problems:
  - fibonacci numbers
  - edit distance
  - knapsack

next time: *more* dynamic programming

TOC