

## Exponentiation

```
int exponentiate( int a, int exponent ) {
    int result = 1;
    for( int i = 1; i <= exponent; i++ ) {
        result *= a;
    }
    return result;
}
```

- Runtime:  $\Theta(n)$ 
  - if multiplication takes constant time!
- Risk of overflow:  $15^{17} > 2^{64}$
- Runtime if size considered:  $\Theta(\text{size}(a)^{\lg^3 n^2})$ 
  - using divide-and-conquer multiplication
- Runtime of `exponentiateDC()`:  $\Theta(\text{size}(a)^{\lg^3 n \lg^3})$

© 2005-06. 324-1-88

## Exponentiation

- Improved exponentiation algorithm
  - observation:  $a^n = (a^{n/2})^2$  if  $n$  is even
- Example:
  - $a^{29} = a a^{28} = a (a^{14})^2 = a((a^7)^2)^2 = \dots = a((a(aa^2)^2)^2)^2$
  - 7 multiplications instead of 28!!!
- Algorithm:

```
int exponentiateDC( int a, int exponent ) {
    if( exponent == 1 ) {
        return a;
    }
    if( exponent % 2 == 0 ) {
        int i = exponentiateDC( a, exponent / 2 );
        return i * i;
    }
    return a * exponentiateDC( a, exponent - 1 );
}
```

© 2005-06. 324-1-89

## Dynamic Programming

- Idea: create a data structure to hold intermediate values
  - e.g. a list, or a table
  - fill it up as you go along
  - bottom up
    - divide-and-conquer: top-down
- Simple example: Binomial Coefficient
  - additive formula
    - runtime: more than exponential
  - factorial formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{if } n \geq k \geq 0$$

- runtime:  $O(n \cdot k)$
- can be calculated using Pascal's Triangle
  - runtime: ?
  - dynamic programming!

© 2005-06. 324-1-90

## Making Change, 2nd version

- Greedy algorithm does not always work optimally
  - for example, not if denominations are too close together
  - coins: 1,4,6
    - make change for 8
      - non-optimal result
- Idea
  - set up table
    - one row (i) for each denomination
    - one column (j) for each amount 0...n
    - each entry: minimum number of coins for amount using only smaller denominations
    - $c[i,j] = \min( c[ i-1, j ], 1 + c[ i, j - \text{denomination}(i) ] )$

© 2005-06. 324-1-91

## Making Change: Example

i \ j	0c	1c	2c	3c	4c	5c	6c	7c	8c
1c	0	1 1 @ 1c	2 2 @ 1c	3 3 @ 1c	4 4 @ 1c	5 5 @ 1c	6 6 @ 1c	7 7 @ 1c	8 8 @ 1c
4c	0	1 1 @ 1c	2 2 @ 1c	3 3 @ 1c	1 1 @ 4c	2 1 @ 4c 1 @ 1c	3 1 @ 4c 2 @ 1c	4 1 @ 4c 3 @ 1c	2 2 @ 4c
6c	0	1 1 @ 1c	2 2 @ 1c	3 3 @ 1c	1 1 @ 4c	2 1 @ 4c 1 @ 1c	1 1 @ 6c	2 1 @ 6c 1 @ 1c	2 2 @ 4c

i = largest denomination of coins used  
j = amount to make change for

Problem: make change for 8 cents

© 2005-06. 324-1-92

## Principle of Optimality

- "In an optimal sequence of decisions, each subsequence must be optimal."
  - or: if an intermediate result is optimal, you never need to recompute it
- Requirement for dynamic programming
  - Making change: each entry in the table is optimal for the denominations considered
- Examples
  - finding shortest distance between cities
    - adheres to principle
  - finding cheapest airline tickets between two cities
    - does not adhere to principle

© 2005-06. 324-1-93

## Knapsack Problem

- Knapsack = backpack
- Problem:
  - a knapsack can hold a maximum weight
  - you have a number of objects you want to store
  - each has a weight and a value
  - goal: maximize value without going over weight limit
- Greedy algorithm
  - select objects by value/weight ratio
  - works only if you can split up objects...
- Recursive algorithm
  - exponential
  - compare: recursive Fibonacci algorithm

© 2005-06. 324-1-94

## Knapsack: Dyn. Prog. Algorithm

- Table
  - rows (i): last object considered
  - column (j): weight limits from 0...maxWeight
  - cells: max value
- Formula
$$\text{value}[i, j] = \max(\text{value}[i-1, j], \text{value}[i-1, j-\text{weight}(i)] + \text{value}(i))$$
  - same as Making Change except we add up value instead of number of coins
- Algorithm:
  - create table row by row
  - $O(n * \text{maxWeight})$
  - what to do if maxWeight is very large or a fraction?

© 2005-06. 324-1-95

## Sequence Alignment

- Needleman-Wunsch algorithm
- Problem: find out how many changes needed to transform two strings so they are equal
  - possible steps:
    - insert a wildcard (“gap”)
    - change one character into a wildcard
- Example: “abbac” versus “abcbc”
  - 1. step:        abbac -> ab\_bac (insert wildcard)
  - 2. step:        abcbc -> abcb\_c (insert wildcard)
- Used very often in biology
  - “*the* dynamic programming algorithm”
  - alignment of DNA sequences
  - to determine genetic relationship
  - the closer their DNA sequences are: the closer are the two plants or animals related
  - DNA sequence: string of a four-letter alphabet {A,C,G,T}

© 2005-06. 324-1-97

## Sequence Alignment

- Create table
  - rows: characters of sequence 1
  - columns: characters of sequence 2
- In each cell:
  - take smallest of:
    - take left cell and add 1
      - insert wildcard into left sequence
    - take above cell and add 1
      - insert wildcard into top sequence
    - take above-and-left cell and add 1
      - if the two characters don't match
      - change character into wildcard
    - take above-and-left cell and add 0
      - if the two characters match
- In red: path to optimal solution
  - there may be more than one

© 2005-06. 324-1-98

## Sequence Alignment Example

		<b>A</b>	<b>C</b>	<b>T</b>	<b>C</b>	<b>G</b>
	<b>0</b>	1 *	2 **	3 ***	4 ****	5 *****
<b>A</b>	1 *	<b>0</b> A	1 A*	2 A**	3 A***	4 A****
<b>C</b>	2 **	1 A*	<b>0</b> AC	1 AC*	2 AC**	3 AC***
<b>A</b>	3 ***	2 A**	<b>1</b> AC*	1 AC*	2 AC**	3 AC***
<b>G</b>	4 ****	3 A***	<b>2</b> AC**	2 AC**	2 AC**	2 AC**G
<b>T</b>	5 *****	4 A****	3 AC***	<b>2</b> AC**T	3 AC***	3 AC***
<b>A</b>	6 *****	5 A*****	4 AC****	3 AC**T*	<b>3</b> AC**T*	4 AC**T**
<b>G</b>	7 *****	6 A*****	5 AC*****	4 AC**T**	4 AC**T**	<b>3</b> AC**T*G