CS 473: Algorithms

Chandra Chekuri
chekuri@cs.uiuc.edu
3228 Siebel Center

University of Illinois, Urbana-Champaign

Fall 2008

Part I

Introduction to Dynamic Programming

Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

- reduce problem to a *smaller* instance of *itself*
- self-reduction

Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

- reduce problem to a *smaller* instance of *itself*
- self-reduction

- Problem instance of size n is reduced to one or more instances of size $n-1$ or less.
- For termination, problem instances of small size are solved by some other method as *base cases*

Recursion in Algorithm Design

- Tail Recursion: problem reduced to a *single* recursive call after some work. Easy to convert algorithm into iterative or greedy algorithms. Examples: Huffman codes, MST algorithms, etc.
- Divide and Conquer: problem reduced to multiple *independent* sub-problems that are solved separately. Conquer step puts together solution for bigger problem.
- Dynamic Programming: problem reduced to multiple (typically) *dependent or overlapping* sub-problems. Use *memoization* to avoid recomputation of common solutions leading to *iterative bottom-up* algorithm.

Fibonacci Numbers

Fibonacci numbers defined by recurrence:

$$F(n) = F(n-1) + F(n-2) \text{ and } F(0) = 0, F(1) = 1.$$

These numbers have many interesting and amazing properties. A journal *The Fibonacci Quarterly*!

- $F(n) = (\phi^n - (1-\phi)^n)/\sqrt{5}$ where ϕ is the golden ratio $(1+\sqrt{5})/2 \simeq 1.618$.
- $\lim_{n\to\infty} F(n+1)/F(n) = \phi$

Fibonacci Numbers

Fibonacci numbers defined by recurrence:

$$F(n) = F(n-1) + F(n-2) \text{ and } F(0) = 0, F(1) = 1.$$

These numbers have many interesting and amazing properties.
A journal *The Fibonacci Quarterly*!

- $F(n) = (\phi^n - (1-\phi)^n)/\sqrt{5}$ where ϕ is the golden ratio $(1+\sqrt{5})/2 \simeq 1.618$.
- $\lim_{n\to\infty} F(n+1)/F(n) = \phi$

Question: Given n, compute $F(n)$.

Recursive Algorithm for Fibonacci Numbers

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

Recursive Algorithm for Fibonacci Numbers

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

Running time? Let $T(n)$ be the number of additions in Fib(n).

Recursive Algorithm for Fibonacci Numbers

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

Running time? Let $T(n)$ be the number of additions in Fib(n).

$$T(n) = T(n-1) + T(n-2) + 1 \text{ and } T(0) = T(1) = 0$$

Roughly same as $F(n)$!

$$T(n) = \Theta(\phi^n)$$

Thus algorithm does exponential in n additions!

Recursive Algorithm for Fibonacci Numbers

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

Running time? Let $T(n)$ be the number of additions in Fib(n).

$$T(n) = T(n-1) + T(n-2) + 1 \text{ and } T(0) = T(1) = 0$$

Roughly same as $F(n)$!

$$T(n) = \Theta(\phi^n)$$

Thus algorithm does exponential in n additions! Can we do better?

An iterative algorithm for Fibonacci numbers

```
Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        F[0] = 0
        F[1] = 1
        for i = 2 to n do
            F[i] = F[i-1] + F[i-2]
        return F[n]
```

An iterative algorithm for Fibonacci numbers

```
Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        F[0] = 0
        F[1] = 1
        for i = 2 to n do
            F[i] = F[i-1] + F[i-2]
        return F[n]
```

What is the running time of the algorithm?

An iterative algorithm for Fibonacci numbers

```
Fib(n):
    if (n = 0)
        return 0
    else if (n = 1)
        return 1
    else
        F[0] = 0
        F[1] = 1
        for i = 2 to n do
            F[i] = F[i-1] + F[i-2]
        return F[n]
```

What is the running time of the algorithm? $O(n)$ additions.

What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value.

What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value. Memoization.

What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value. Memoization.

Dynamic Programming: finding a recursion that can be *effectively/efficiently* memoized

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.

Back to Fibonacci Numbers

Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

Back to Fibonacci Numbers

Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is n and hence input size is $\Theta(\log n)$

Back to Fibonacci Numbers

Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is n and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$! Why?

Back to Fibonacci Numbers

Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is n and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$! Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!

Back to Fibonacci Numbers

Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is n and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$! Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
- Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?

Back to Fibonacci Numbers

Is the iterative algorithm a *polynomial* time algorithm? Does it take $O(n)$ time?

- input is n and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$! Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
- Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?
- Running time of recursive algorithm is $O(n\phi^n)$ but can in fact shown to be $O(\phi^n)$ by being careful. Doubly exponential in input size!