CMPS 3120

Algorithm Analysis

Dr. Chengwei Lei
CEECS
California State University, Bakersfield
Example 1: Recursive evaluation of $n!$

Definition: $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$ for $n \geq 1$ and $0! = 1$

Recursive definition of $n!$: $F(n) = F(n-1) \cdot n$ for $n \geq 1$ and $F(0) = 1$

**Algorithm**

```
F(n)
// Computes n! recursively
// Input: A nonnegative integer n
// Output: The value of n!
if n = 0 return 1
else return F(n - 1) * n
```

Size:
Basic operation:
Recurrence relation:
Solving the recurrence for $M(n)$

$M(n) = M(n-1) + 1, \ M(0) = 0$
Example 2: The Tower of Hanoi Puzzle

Recurrence for number of moves:
Solving recurrence for number of moves

\[ M(n) = 2M(n-1) + 1, \quad M(1) = 1 \]
Tree of calls for the Tower of Hanoi Puzzle

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```

```
1 1 1 1
2 2 ...
```
Example 3: Counting #bits

ALGORITHM  BinRec(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n’s binary representation
if n = 1 return 1
else return BinRec(\lfloor n/2 \rfloor) + 1
Recursive algorithms

- Why? How?
Recursive algorithms

- General idea:
  - Divide a large problem into smaller ones
    - By a constant ratio
    - By a constant or some variable
  - Solve each smaller one recursively or explicitly
  - Combine the solutions of smaller ones to form a solution for the original problem

**Divide and Conquer**
Merge sort
Merging two sorted arrays

Subarray 1
- 20
- 13
- 7
- 2

Subarray 2
- 12
- 11
- 9
- 1
Merging two sorted arrays

Subarray 1

Subarray 2

20
13
7
2

12
11
9
1
Merging two sorted arrays

20 12
13 11
7 9
2 1
Merging two sorted arrays

20 12
13 11
7 9
2 1
Merging two sorted arrays
Merging two sorted arrays

20 12 || 20 12
13 11 || 13 11
 7 9 ||  7 9
 2 1 ||  2  
 1  ||   

16
Merging two sorted arrays

20  12  |  20  12
13  11  |  13  11
  7  9  |   7  9
  2  1  |   2  2
Merging two sorted arrays
Merging two sorted arrays

20 12 20 12 20 12
13 11 13 11 13 11
7 9 7 9 7 9
2 1 2 7 7
1 2 7
Merging two sorted arrays

20 12 || 20 12 || 20 12 || 20 12
13 11 || 13 11 || 13 11 || 13 11
7 9 || 7 9 || 7 9 || 7 9
2 1 || 2 1 || 7 9 || 9
1 2 7

9/25/2018
Merging two sorted arrays

20 12 || 20 12 || 20 12 || 20 12
13 11 || 13 11 || 13 11 || 13 11
7  9  ||  7  9  ||  7  9  ||  13 11
 2 1 ||  2 2  ||  7 2  ||  9 9
Merging two sorted arrays
Merging two sorted arrays

20  12  ||  20  12  ||  20  12  ||  20  12  ||  20  12
13  11  ||  13  11  ||  13  11  ||  13  11  ||  13  11
  7   9  ||   7   9  ||  7   9  ||   7   9  ||   7   9
  2  1  ||  2   2  ||  7   7  ||  9   9  ||  9   9
  1   2  ||  2   7  ||  7   9  ||  9   9  ||  9   9
           ||          ||          ||          ||

9/25/2018
Merging two sorted arrays
Merging two sorted arrays
How to show the correctness of a recursive algorithm?

- By induction:
  - **Base case**: prove it works for small examples
  - **Inductive hypothesis**: assume the solution is correct for all sub-problems
  - **Step**: show that, if the inductive hypothesis is correct, then the algorithm is correct for the original problem.
Correctness of merge sort

**Merge-Sort** $A[1 . . n]$

1. If $n = 1$, done.
2. Recursively sort $A[1 . . \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 . . n]$.
3. “Merge” the 2 sorted lists.

**Proof:**

1. **Base case:** if $n = 1$, the algorithm will return the correct answer because $A[1..1]$ is already sorted.
2. **Inductive hypothesis:** assume that the algorithm correctly sorts $A[1..\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil +1..n]$.
3. **Step:** if $A[1..\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil +1..n]$ are both correctly sorted, the whole array $A[1..\lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil +1..n]$ is sorted after merging.
How to analyze the time-efficiency of a recursive algorithm?

- Express the running time on input of size $n$ as a function of the running time on *smaller* problems
Analyzing merge sort

<table>
<thead>
<tr>
<th>$T(n)$</th>
<th><strong>Merge-Sort</strong> $A[1 \ldots n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(1)$</td>
<td>1. If $n = 1$, done.</td>
</tr>
<tr>
<td>$2T(n/2)$</td>
<td>2. Recursively sort $A[1 \ldots \lceil n/2 \rceil]$ and $A[\lfloor n/2 \rfloor+1 \ldots n]$.</td>
</tr>
<tr>
<td>$f(n)$</td>
<td>3. “Merge” the 2 sorted lists</td>
</tr>
</tbody>
</table>

**Sloppiness:** Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.
Analyzing merge sort

1. **Divide:** Trivial.
2. **Conquer:** Recursively sort 2 subarrays.
3. **Combine:** Merge two sorted subarrays

\[ T(n) = 2T(n/2) + f(n) + \Theta(1) \]

# subproblems
\( f(n) \)
subproblem size
Dividing and Combining

1. What is the time for the base case? **Constant**
2. What is \( f(n) \)?
3. What is the growth order of \( T(n) \)?
Merging two sorted arrays

$\Theta(n)$ time to merge a total of $n$ elements (linear time).
Recurrence for merge sort

\[ T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases} \]

We will usually ignore the base case, assuming it is always a constant (but not 0).
Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.
Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.
Recursion tree for merge sort

Solve \( T(n) = 2T(n/2) + dn \), where \( d > 0 \) is constant.
Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.
Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.
Recursion tree for merge sort

Solve \( T(n) = 2T(n/2) + dn \), where \( d > 0 \) is constant.

\[ h = \log n \]

\[ \Theta(1) \]
Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.

$h = \log n$

$\Theta(1)$
Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.
Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.

$h = \log n$

$\Theta(1)$
Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.
Recursion tree for merge sort

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.

Later we will usually ignore $d$. Total $\Theta(n \log n)$.
Problem of the day

- Write the Pseudo code to compute $b^n$
Problem of the day

How many multiplications do you need to compute $3^{16}$?

$3^{16} = 3 \times 3 \times 3 \ldots \times 3$  \hspace{1cm} \text{Answer: 15}

$3^{16} = 3^8 \times 3^8$

$3^8 = 3^4 \times 3^4$

$3^4 = 3^2 \times 3^2$

$3^2 = 3 \times 3$  \hspace{1cm} \text{Answer: 4}
Pseudo code

```c
int pow (b, n)  // compute \(b^n\)
    m = n >> 1;
    p = pow (b, m);
    p = p * p;
    if (n % 2)
        return p * b;
    else
        return p;
```
Pseudo code

int pow (b, n)  
    m = n >> 1;
    p = pow (b, m);
    p = p * p;
    if (n % 2)
        return p * b;
    else
        return p;

int pow (b, n)  
    m = n >> 1;
    p = pow(b,m) * pow(b,m);
    if (n % 2)
        return p * b;
    else
        return p;
Recurrence for computing power

```c
int pow (b, n)
{
    m = n >> 1;
    p = pow (b, m);
    p = p * p;
    if (n % 2)
        return p * b;
    else
        return p;
}
```

```c
int pow (b, n)
{
    m = n >> 1;
    p = pow (b, m) * pow (b, m);
    if (n % 2)
        return p * b;
    else
        return p;
}
```

T(n) = ?
Recurrence for computing power

int pow (b, n)
  m = n >> 1;
  p = pow (b, m);
  p = p * p;
  if (n % 2)
    return p * b;
  else
    return p;

T(n) = T(n/2)+Θ(1)

int pow (b, n)
  m = n >> 1;
  p=pow(b,m)*pow(b, m);
  if (n % 2)
    return p * b;
  else
    return p;

T(n) = 2T(n/2)+Θ(1)
Recurrence for computing power

int pow (b, n)
    m = n >> 1;
    p = pow (b, m);
    p = p * p;
    if (n % 2)
        return p * b;
    else
        return p;

T(n) = T(n/2)+Θ(1)

Which algorithm is more efficient asymptotically?
Time complexity for Alg 1

Solve $T(n) = T(n/2) + 1$

$= T(n/4) + 1 + 1$
$= T(n/8) + 1 + 1 + 1$
$= T(1) + 1 + 1 + ... + 1$

$= \Theta (\log(n))$

Iteration method
Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$. 
Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$. 

$T(n)$
Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$. 

![Diagram showing the recurrence relation $T(n) = 2T(n/2) + 1$]

- $T(n/2)$
- $T(n/2)$
- $1$
Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$. 
Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$. 

\[ \Theta(1) \]
Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$. 

$h = \log n$ 

$\Theta(1)$
Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.

$h = \log n$
Time complexity for Alg2

Solve \( T(n) = 2T(n/2) + 1 \).

\[ h = \log n \]

\[ \Theta(1) \]
Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.

$h = \log n$,

$\Theta(1)$
Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$. 

$h = \log n$ 

$\Theta(1)$ 

$\#leaves = n$ 

$\Theta(n)$
Time complexity for Alg2

Solve $T(n) = 2T(n/2) + 1$.

$h = \log n$

$\Theta(1)$

#leaves $= n$

Total $\Theta(n)$

$\Theta(1)$

$\Theta(n)$
More iteration method examples

- \( T(n) = T(n-1) + 1 \)
  
  \( = T(n-2) + 1 + 1 \)
  
  \( = T(n-3) + 1 + 1 + 1 \)
  
  \( = T(1) + 1 + 1 + \ldots + 1 \)
  
  \( = \Theta(n) \quad n - 1 \)
More iteration method examples

\[ T(n) = T(n-1) + n \]
\[ = T(n-2) + (n-1) + n \]
\[ = T(n-3) + (n-2) + (n-1) + n \]
\[ = T(1) + 2 + 3 + \ldots + n \]
\[ = \Theta(n^2) \]
More recursion tree examples

- \( T(n) = 3T(n/3) + n \)
- \( T(n) = T(n/3) + T(2n/3) + n \)
- \( T(n) = 2T(n/4) + n \)
- \( T(n) = 2T(n/4) + n^2 \)
- \( T(n) = T(n-2) + n \)
- \( T(n) = T(n-2) + 1 \)