Analyzing Recursive Algorithms

Ref: Section 2.3 and Chapter 4 of text.

Example 1: Factorial function. (Assume that \( n \geq 1 \).

Pseudocode:
Fact (n)
  if (n = 1)
    return 1 /* Step (a) */
  else
    return n*Fact(n-1) /* Step (b) */

Estimating running time:
• \( T(n) \): Running time when input is \( n \).
• \( c \) (constant): Time for the comparison in Step (a).
• \( c_1 \) (constant): Time for the comparison and return in Step (a).
• \( c_2 \) (constant): Time for the multiplication and return in Step (b).

Recurrence for \( T(n) \):
\[
T(1) = c_1 \\
T(n) = c + T(n - 1) + c_2, \ n \geq 2
\]

Equivalently,
\[
T(1) = c_1 \\
T(n) = c' + T(n - 1), \ n \geq 2
\]

We can solve for \( T(n) \) by the iteration method.
\[
T(n) = c' + T(n - 1) \\
= c' + c' + T(n - 2) \\
= 2c' + T(n - 2) \\
\vdots \\
= (n - 1)c' + T(1) \\
= (n - 1)c' + c_1 \\
= O(n).
\]

Notes:
1. Iterative version, which also runs in \( O(n) \) time, is likely to run faster in practice.
2. When \( n \) gets large, multiplication time may not be \( O(1) \).
**Example 2:** Fibonacci numbers.

\[
\begin{align*}
F(0) &= 0 \\
F(1) &= 1 \\
F(n) &= F(n-1) + F(n-2), \quad n \geq 2
\end{align*}
\]

**Pseudocode:**

```
FI(n)
  if (n <= 1)
    return n
  else
    return FI(n-1) + FI(n-2)
```

**Estimating running time:** Let \( T(n) \) be the running time when input is \( n \).

\[
\begin{align*}
T(0) &= c \\
T(1) &= c \\
T(n) &= T(n-1) + T(n-2) + c_1, \quad n \geq 2.
\end{align*}
\]

where \( c \) and \( c_1 \) are constants.

**Exercise:** Assume that \( c = c_1 = 1 \). Use induction on \( n \) to prove that \( T(n) \geq (3/2)^{n-1} \).

**Conclusion:** Recursion is a BAD way of computing \( F(n) \).

**Reason:** The same value gets computed many times.

**Note:** Easy to devise an \( O(n) \) iterative algorithm that stores values in an array.

**Divide-and-Conquer technique:**

- Divide step: Divide the problem into smaller subproblems.
- Conquer step: Solve the subproblems recursively. (Recursion stops when the subproblem size is small.)
- Combine step: Combine the subproblem solutions to obtain a solution to the original problem.

**Example 1:** Finding the maximum value in array \( A[1 \ldots n] \).

**Idea:**

```
1   n/2   n/2+1   n
A

Left Half  |  Right Half

1 3-3
```

3-4
Max($A$) = Larger of Max(Left Half) and Max(Right Half).

Steps:
- Divide step: Divide the array into two equal halves.
- Conquer step: Recursively compute the maximum value of left half (say, $\text{max}l$) and right half (say, $\text{max}r$). Recursion stops when each subarray is of size 1.
- Combine step: Return the larger of $\text{max}l$ and $\text{max}r$.

Pseudocode: Easy exercise.

Estimating running time:
- Assume: $n = 2^r$ for some positive integer $r$.
- $T(n)$: Running time for an array of size $n$.

\[
T(1) = c_1 \\
T(n) = c_2 + T(n/2) + T(n/2) + c_3, \ n \geq 2.
\]

where $c_1$, $c_2$ (time for divide) and $c_3$ (time for combine) are constants.

Equivalently,
\[
T(1) = c_1 \\
T(n) = 2T(n/2) + c_4, \ n \geq 2.
\]

where $c_4 = c_2 + c_3$ is also a constant.

Solution: (to be done in class)
\[
T(n) = c_1 n + c_4(n - 1).
\]

In other words, $T(n) = O(n)$.

Things to remember:
- Divide step usually breaks the original problem into two subproblems of equal size (half the original size).
- Conquer step solves the same problem recursively on the subproblems. (Where recursion ends depends on the problem.)
- Usually, the most creative parts of the algorithm are in the Combine step.

Example 2: Finding the maximum and minimum values in array $A[1 \ldots n]$.

Measure: The number of comparisons used.
Algorithm Easy:
1. Simple loop to find max \((n - 1\) comparisons).
2. Remove max and use another loop to find min 
   \((n - 2\) comparisons).
Total = \(2n - 3\) comparisons.

Divide-and-Conquer algorithm:
- Divide step: Divide the array into two equal halves.
- Conquer step:
  - Recursively compute the maximum and minimum values of left half (say, \(max_l\) and 
    \(min_l\)) and right half (say, \(max_r\) and \(min_r\)).
  - Recursion stops when each subarray is of size 2. (Then one comparison suffices to 
    get max and min.)
- Combine step:
  - Max for the array is the larger of \(max_l\) and \(max_r\).
  - Min for the array is the smaller of \(min_l\) and \(min_r\).

Pseudocode: See Handout 3.1.

Number of comparisons:
- Assume: \(n = 2^r\) for some positive integer \(r\).
- \(T(n)\) : No. of comparisons for array of size \(n\).

Recurrence:
\[
T(2) = 1 \\
T(n) = 2T(n/2) + 2, \quad n > 2.
\]

Solution: (to be done in class)
\[
T(n) = \frac{3}{2}n - 2.
\]

Note: D-and-C algorithm uses about \(n/2\) fewer comparisons than Algorithm Easy.

Example 3: Merge Sort. (Sort array \(A[1 .. n]\) into increasing order.)
Digression: Merging two sorted arrays:

Given: Two sorted arrays $X$ (size: $n_1$) and $Y$ (size: $n_2$)

Required: Merge the two arrays into a single sorted array $Z$.

Algorithm Idea:
- Scan $X$ and $Y$ from beginning to end, writing the smaller value to $Z$.
- When one of $X$ and $Y$ is done, copy the values left in the other array into $Z$.

Running time of Merge: $O(n_1 + n_2)$.

Merge Sort algorithm:
- Divide step: Divide the array into two equal halves.
- Conquer step:
  - Recursively sort the left and right halves of the array.
  - Recursion stops when each subarray is of size 1. (In that case, the subarray is already sorted.)

- Combine step: Merge the two sorted halves.

Pseudocode: (from page 32 of text)

```
MERGE-SORT (A, p, r)
if (p < r)
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Note: The call $\text{MERGE}(A, p, q, r)$ merges the subarrays $A[p .. q]$ and $A[q+1 .. r]$ and the result is stored in $A[p .. r]$.

Running time of Merge Sort:
- Assume: $n = 2^r$ for some positive integer $r$.
- $T(n)$ : Time for an array of size $n$.

Recurrence: ($c_1, c_2 : \text{constants}$)

$$
T(1) = c_1 \\
T(n) \leq 2T(n/2) + c_2n, \quad n \geq 2.
$$

Solution: (to be done in class)

$T(n) = O(n \log n)$.
Another explanation for running time:

- Recall: Time to merge two arrays of sizes $n_1$ and $n_2$ is $\leq c_1(n_1 + n_2)$ and each divide operation takes $c_2$ (constant) time.
- Running time = Time for all merge operations + Time for all divide operations.
- In the recursion tree for Merge Sort, the number of levels is $\log_2 n$.

**Time for merge operations:**

<table>
<thead>
<tr>
<th>Level</th>
<th>Time per Merge</th>
<th>#Merges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$2c_1$</td>
<td>$n/2$</td>
</tr>
<tr>
<td>2</td>
<td>$2^2c_1$</td>
<td>$n/2^2$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$2^{\log_2 n}c_1$</td>
<td>$n/2^{\log_2 n}$</td>
</tr>
</tbody>
</table>

**General:** At level $i$, there are $n/2^i$ merges and each merge uses $c_1 2^i$ time ($i = 1, 2, \ldots, \log_2 n$).

Total merge time $= \sum_{i=1}^{\log_2 n} \left( \frac{n}{2^i} c_1 2^i \right)$

$= c_1 n \log_2 n$.

**Time for divide operations:**

<table>
<thead>
<tr>
<th>Level</th>
<th>Time per Divide</th>
<th>#Divides</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$c_2$</td>
<td>$n/2$</td>
</tr>
<tr>
<td>2</td>
<td>$c_2$</td>
<td>$n/2^2$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$c_2$</td>
<td>$n/2^{\log_2 n}$</td>
</tr>
</tbody>
</table>

**General:** At level $i$, there are $n/2^i$ divide operations. Each divide uses $c_2$ time ($i = 1, 2, \ldots, \log_2 n$).

Total divide time $= \sum_{i=1}^{\log_2 n} \left( \frac{n}{2^i} c_2 \right)$

$= c_2 n \sum_{i=1}^{\log_2 n} \frac{1}{2^i}$

$< c_2 n \sum_{i=1}^{\infty} \frac{1}{2^i}$

$< c_2 n \times 2 = c_2 n$.

Running time $< c_1 n \log_2 n + c_2 n = O(n \log n)$. 

3-11