Lecture 2: Divide-and-Conquer and Growth of Functions

(CLRS 2.3,3)

January 15, 2002

1 Designing Good Algorithms: Divide and Conquer/Mergesort

1.1 Divide-and-conquer

- Last time we discussed insertion sort
  - We introduced RAM model of computation and discussed its limitations.
  - We analyzed insertion sort in the RAM model
    * Best-case $k_1n - k_2$.
    * Worst-case (and average case) $k_3n^2 + k_4 - k_5$
  - We discussed how we are normally only interested in growth of running time:
    * Best-case linear in $n$ ($\sim n$), worst-case quadratic in $n$ ($\sim n^2$).

- Can we design better than $n^2$ sorting algorithm?

- We will do so using one of the most powerful algorithm design techniques.

<table>
<thead>
<tr>
<th>Divide and Conquer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>To Solve P:</strong></td>
</tr>
<tr>
<td>1. <em>Divide</em> P into smaller problems $P_1, P_2, P_3, \ldots, P_k$.</td>
</tr>
<tr>
<td>2. <em>Conquer</em> by solving the (smaller) subproblems recursively.</td>
</tr>
<tr>
<td>3. <em>Combine</em> solutions to $P_1, P_2, \ldots, P_k$ into solution for P.</td>
</tr>
</tbody>
</table>

1.2 Merge-Sort

- Using divide-and-conquer, we can obtain a merge-sort algorithm.
  - Divide: Divide $n$ elements into two subsequences of $n/2$ elements each.
  - Conquer: Sort the two subsequences recursively.
  - Combine: Merge the two sorted subsequences.

- Assume we have procedure $\text{Merge}(A, p, q, r)$ which merges sorted $A[p..q]$ with sorted $A[q+1..r]$ in $(r - p)$ time.
• We can sort $A[p...r]$ as follows (initially $p=1$ and $r=n$):

```
Merge Sort(A,p,r)
  If $p < r$ then
    $q = \lfloor (p + r)/2 \rfloor$
    MergeSort(A,p,q)
    MergeSort(A,q+1,r)
    Merge(A,p,q,r)
```

Example:

```
5 2 4 6 1 3 2 6
5 2 4 6 1 3 2 6
5 2 4 6 1 3 2 6
5 2 4 6 1 3 2 6
1 2 2 3 4 5 6 6
1 2 2 3 4 5 6 6
1 2 2 3 4 5 6 6
1 2 2 3 4 5 6 6
```

1.3 Correctness

• Induction on $n$
  
  – Easy assuming $\text{Merge()}$ is correct!
1.4 Analysis

- To simplify things, let us assume that \( n \) is a power of 2, i.e. \( n = 2^k \) for some \( k \).
- Running time of the procedure can be analyzed using a recurrence equation/rela-
tion.

\[
T(n) \leq c_1 + T(n/2) + T(n/2) + c_2 n \\
\leq 2T(n/2) + c_3 n
\]

\[\downarrow\]

\[T(n) \leq c_1 n \log_2 n \text{ as we will see later.}\]

- We can also get \( n \log_2 n \) bound by noticing that the recursion tree has depth \( \log_2 n \) and that linear time is spent on each level.
- Note:
  - We really have \( T(n) = c_4 \) if \( n = 1 \)
  - If \( n \neq 2^k \) things can be complicated (We will often assume \( n = 2^k \) to avoid complicated cases).

1.5 Log’s

- Base 2 logarithm comes up all the time (from now on we will always mean \( \log_2 n \) when we write \( \log n \)).
  - Number of times we can divide \( n \) by 2 to get to 1 or less.
  - Number of bits in binary representation of \( n \).
  - Inverse function of \( 2^n = 2 \cdot 2 \cdot 2 \cdots 2 \) (\( n \) times).
  - Way of doing multiplication by addition: \( \log(ab) = \log(a) + \log(b) \)

- Note:
  - \( \log_a n = \frac{\log_b n}{\log_b a} \)
  - \( \log n << \sqrt{n} << n \)

1.6 Algorithms matter!

- Sort 10 million integers on
  - 1 GHz computer (1000 million instructions per second) using \( 2n^2 \) algorithm.
  - 100 MHz computer (100 million instructions per second) using \( 50n \log n \) algorithm.

- Supercomputer: \[
\frac{2 \cdot (10^7)^2 \text{ inst.}}{10^9 \text{ inst. per second}} = 200000 \text{ seconds} \approx 55 \text{ hours}.
\]

- Personal computer: \[
\frac{50 \cdot 10^7 \cdot \log 10^7 \text{ inst.}}{10^8 \text{ inst. per second}} < \frac{50 \cdot 10^7 \cdot 7.3}{10^8} = 5 \cdot 7 \cdot 3 = 105 \text{ seconds}.
\]
2 Asymptotic Growth

- In the insertion-sort example we discussed that when analyzing algorithms we are
  - interested in worst-case running time as function of input size $n$
  - not interested in exact constants in bound
  - not interested in lower order terms

- A good reason for not caring about constants and lower order terms is that the RAM model
  is not completely realistic anyway (not all operations cost the same)

\[ \downarrow \]

- We want to express rate of growth of standard functions:
  - the leading term with respect to $n$
  - ignoring constants in front of it

\[
\begin{align*}
  k_1n + k_2 & \sim n \\
  k_1n \log n & \sim n \log n \\
  k_1n^2 + k_2n + k_3 & \sim n^2
\end{align*}
\]

- We also want to formalize e.g. that a $n \log n$ algorithms is better than a $n^2$ algorithm.

\[ \downarrow \]

- $O$-notation (Big-$O$)
  - you have probably all seen it intuitively defined but we will now define it more carefully.

2.1 $O$-notation (Big-$O$)

\[
O(g(n)) = \{ f(n) : \exists c, n_0 > 0 \text{ such that } f(n) \leq cg(n) \ \forall n \geq n_0 \}
\]

- $O(\cdot)$ is used to asymptotically upper bound a function.
- $O(\cdot)$ is used to bound worst-case running time.
• Examples:
  - \( \frac{1}{3}n^2 - 3n \in O(n^2) \) because \( \frac{1}{3}n^2 - 3n \leq cn^2 \) if \( c \geq \frac{1}{3} - \frac{3}{n} \) which holds for \( c = \frac{1}{3} \) and \( n > 1 \).
  - \( k_1n^2 + k_2n + k_3 \in O(n^2) \) because \( k_1n^2 + k_2n + k_3 < (k_1 + |k_2| + |k_3|)n^2 \) and for \( c > k_1 + |k_2| + |k_3| \) and \( n \geq 1 \), \( k_1n^2 + k_2n + k_3 < cn^2 \).
  - \( k_1n^2 + k_2n + k_3 \in O(n^3) \) as \( k_1n^2 + k_2n + k_3 < (k_1 + k_2 + k_3)n^3 \) (Upper bound!).

• Note:
  - When we say “the running time is \( O(n^2) \)” we mean that the worst-case running time is \( O(n^2) \) — best case might be better.
  - Use of \( O \)-notation often makes it much easier to analyze algorithms; we can easily prove the \( O(n^2) \) insertion-sort time bound by saying that both loops run in \( O(n) \) time.
  - We often abuse the notation a little:
    * We often write \( f(n) = O(g(n)) \) instead of \( f(n) \in O(g(n)) \).
    * We often use \( O(n) \) in equations: e.g. \( 2n^2 + 3n + 1 = 2n^2 + O(n) \) (meaning that \( 2n^2 + 3n + 1 = 2n^2 + f(n) \) where \( f(n) \) is some function in \( O(n) \)).
    * We use \( O(1) \) to denote constant time.

2.2 \( \Omega \)-notation (big-Omega)

\[ \Omega(g(n)) = \{ f(n) : \exists c, n_0 > 0 \text{ such that } cg(n) \leq f(n) \forall n \geq n_0 \} \]

• \( \Omega(\cdot) \) is used to asymptotically lower bound a function.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{image}
\caption{Graph of functions \( f(n) \), \( cg(n) \) for \( \Omega(g(n)) \).}
\end{figure}

• Examples:
  - \( \frac{1}{3}n^2 - 3n = \Omega(n^2) \) because \( \frac{1}{3}n^2 - 3n \geq cn^2 \) if \( c \leq \frac{1}{3} - \frac{3}{n} \) which is true if \( c = \frac{1}{6} \) and \( n > 18 \).
  - \( k_1n^2 + k_2n + k_3 = \Omega(n^2) \).
  - \( k_1n^2 + k_2n + k_3 = \Omega(n) \) (lower bound!)
• Note:
  – When we say “the running time is $\Omega(n^2)$”, we mean that the best case running time is $\Omega(n^2)$ — the worst case might be worse.

• Insertion-sort:
  – Best case: $\Omega(n)$
  – Worst case: $O(n^2)$
  – We can also say that the worst case running time is $\Omega(n^2)$ \implies worst case running time is “precisely” $n^2$.

2.3 $\Theta$-notation (Big-Theta)

$\Theta(g(n)) = \{ f(n) : \exists c_1, c_2, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \ \forall n \geq n_0 \}$

• $\Theta(\cdot)$ is used to asymptotically tight bound a function.

$$
f(n) = \Theta(g(n)) \text{ if and only if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))
$$

• Examples:
  – $k_1 n^2 + k_2 n + k_3 = \Theta(n^2)$
  – worst case running time of insertion-sort is $\Theta(n^2)$
  – $6n \log n + \sqrt{n \log^2 n} = \Theta(n \log n)$:
    * We need to find $n_0, c_1, c_2$ such that $c_1 n \log n \leq 6n \log n + \sqrt{n \log^2 n} \leq c_2 n \log n$ for $n > n_0$
      $c_1 n \log n \leq 6n \log n + \sqrt{n \log^2 n} \Rightarrow c_1 \leq 6 + \frac{\log n}{\sqrt{n}}$. Ok if we choose $c_1 = 6$ and $n_0 = 1$.
      $6n \log n + \sqrt{n \log^2 n} \leq c_2 n \log n \Rightarrow 6 + \frac{\log n}{\sqrt{n}} \leq c_2$. Is it ok to choose $c_2 = 7$? Yes, $
      \log n \leq \sqrt{n}$ if $n \geq 2$.
    * So $c_1 = 6$, $c_2 = 7$ and $n_0 = 2$ works.


• Note:
  – We often think of \( f(n) = O(g(n)) \) as corresponding to \( f(n) \leq g(n) \).
  – Similarly, \( f(n) = \Theta(g(n)) \) corresponds to \( f(n) = g(n) \)
  – Similarly, \( f(n) = \Omega(g(n)) \) corresponds to \( f(n) \geq g(n) \)
  – One can also define \( o \) and \( \omega \)
    
    * \( f(n) = o(g(n)) \) corresponds to \( f(n) < g(n) \)
    
    * \( f(n) = \omega(g(n)) \) corresponds to \( f(n) > g(n) \)

2.4 Growth rate of standard functions

• Book introduces standard functions in section 2.2 (we will introduce them as we need them):
  
  – Polynomial of degree \( d \): \( p(n) = \sum_{i=1}^{d} a_i \cdot n^i \) where \( a_1, a_2, \ldots, a_d \) are constants (and \( a_d > 0 \)). \( p(n) = \Theta(n^d) \)
  
  • “Growth order”: \( \log \log n, \log n, \sqrt{n}, n, n \log \log n, n \log n, n \log^2 n, n^2, n^3, 2^n \)
    
    – Growth rate of polynomials versus exponentials: \( \lim_{n \to \infty} \frac{n^b}{a^n} = 0 \).