CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- HW9 in already
- HW10 out soon, due Monday, Mar. 16
- Reading (as noted on HW): now in Ch 4, pages 65-67, 88-93
- HW6 back today
- Soln set HW8 out today; HW8 back next week
  - In general, soln sets may be worth looking at, even if you got full credit on the exercises
- Project1 assignment out, due by 8:00am Tuesday, 3/17
  - Note the extended deadline
  - See “Project and Presentation Assignments” on course website
  - Teams for the project have been randomly assigned and are posted on the website
  - Auxiliary files also available for download from projects webpage
Algorithm Correctness Analysis: Loop Invariants

- To prove algorithms correct, can use loop invariants to analyze loops
  - Loop invariants are properties of loops that are true each time through the loop
    - Initialization: Property is true before the first iteration
    - Maintenance: If a property is true before an iteration, it is true (after that iteration) before the next iteration
    - Termination: When the loop terminates, the property (or the violation of the property) is useful in showing algorithm correctness

See CLRS 2.1

Another example: Bubble Sort

- (Yes, bubble sort is the actual name of this sorting algorithm)
- In pseudocode:

```
BubbleSort(A)
    1. for i = 1 to A.length − 1
    2.     for j = A.length downto i + 1
```

- How do we argue correctness (i.e., that it sorts A in non-decreasing order)?
  - What might the loop invariant be for the outer loop?
Another example: Bubble Sort

- Loop invariant for outer loop:

```
Subarray A[1..i-1] consists of the i-1 smallest values of A, in sorted order, and A[i..n] consists of the remaining values of A (no constraint on order)
```

```
BubbleSort(A)
1. for i = 1 to A.length - 1
2. for j = A.length downto i+1
```

- Is the invariant true at initialization (i = 1)?
- Is the invariant maintained by each iteration of the loop—i.e., is it true at the end of an iteration if we assume it’s true at the beginning?
- What does the invariant show is true at termination of the loop? How does that help us show the algorithm overall is correct?

**Sorting Problem**

Input: Sequence of numbers \(<a_1, ..., a_n>\)

Output: Permutation (reordering) \(<b_1, ..., b_n>\) of the input sequence (perhaps leaving them unchanged) such that \(b_1 \leq b_2 \leq ... \leq b_n\)
Using the $\theta$, $\Omega$ Definitions

- Definition: $\theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)\}$
- Definition: $\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0, 0 \leq c * g(n) \leq f(n)\}$
- Is each of the below statements true?
  1. $100n + 5 = \theta(n^2)$
  2. $100n + 5 = \Omega(n^2)$
  3. $n^2/2 - 3n = \theta(n^2)$
  4. $n^2/2 - 3n = \Omega(n^2)$
  5. $100n^2 = \theta(n^3)$
  6. $0.01n^3 = \Omega(n^2)$
  7. $2^n = \theta(2^n)$
  8. $2^n = \Omega(2^n)$

Conventions: Order of Growth
(to within a constant multiple)

- Two different levels of detail can be useful with asymptotic complexity:
  - Formal definitions and detailed explanations
  - Informal, high-level understanding and explanations
- When informally talking about asymptotic complexity, we often talk about
  the order of growth of runtime functions, to within a constant multiple

- Some practice with the informal approach:

  For each of the following pairs of functions, is the order of growth of the
  first function lower, the same, or higher (to within a constant multiple)
  than the order of growth of the second function?

  1. $n(n+1)$ and $2000n^2$
  2. $100n^3$ and $0.01n^3$
  3. $\ln n$ and $\ln n$ (that’s log base 2 and natural log, respectively)
  4. $(n+1)!$ and $n!$
Conventions: Order of Growth (to within a constant multiple)

- Two different levels of detail can be useful with asymptotic complexity:
  - Formal definitions and detailed explanations
  - Informal, high-level understanding and explanations
- When informally talking about asymptotic complexity, we often talk about the order of growth of runtime functions, to within a constant multiple

Some practice with the informal approach:

**True or False (and give a brief explanation):**

1. \(\frac{n(n+1)}{2} \in O(n^3)\)
2. \(\frac{n(n+1)}{2} \in O(n^2)\)
3. \(\frac{n(n+1)}{2} \in \Theta(n^3)\)
4. \(\frac{n(n+1)}{2} \in \Omega(n)\)

Recall the constant multiple(s) in formal definitions of asymptotic complexity...

For assignments given from here on out in CS375, unless specified otherwise, feel free to use the informal, high-level approach.
Mergesort
A Reminder, and Pseudocode

- Mergesort, from CLRS:

```
MergeSort(A, p, r)
if p < r
    q = (p + r)/2
    MergeSort(A, p, q)
    MergeSort(A, q+1, r)
    Merge(A, p, q, r)
```

What technique could we use to write down a formula for the time complexity of Mergesort, or for recursive algorithms in general?
Divide and Conquer and Recurrences

- Mergesort is a canonical example of the divide and conquer strategy for problem solving and algorithm design.

**Review: Divide and conquer—recursive problem solving**
- **Divide** into smaller sub-problems
- **Conquer** by recursion for larger sub-problems (recursive case) or straightforward solution (base case)
- **Combine** sub-problem solutions into full solution

- Complexity of divide and conquer methods often given in terms of recurrence relations (or recurrences, for short)
  - E.g., for mergesort: \( T(n) = \begin{cases} \theta(1) & \text{if } n = 1; \\ 2T(n/2) + \theta(n) & \text{if } n > 1 \end{cases} \)

Recurrences, appropriately enough, will keep showing up throughout CS375!

Solving Recurrences

- Common techniques for solving recurrences—i.e., getting \( \theta \) or \( O \) bounds on the solution:
  - **Unwinding** (or **backward substitution**): “Unroll” the recurrence until it reaches a base case, then count / analyze the cost represented
  - **Recursion-tree method**: Represent costs as nodes in a tree and analyze total cost
  - **Master method**: Solve recurrences of the form
    \[ T(n) = a*T(n/b) + f(n) \]
Unwinding

• An example: Solve \( T(n) = 2T(n/2) + n \)

| What information is missing from this recurrence, which we will need to be able to solve it? |

• *Unwind* the recurrence by plugging in the definition on successively smaller arguments:
  - From the definition, \( T(n) = 2T(n/2) + n \)
  - By that same definition, \( T(n/2) = 2T(n/4) + n/2 \)
  - So, by plugging that in: \( T(n) = 2[2T(n/4) + n/2] + n \)
  - What would the next step(s) be in this unwinding process?
  - Where would it stop?

This name may make it sound more relaxing than it actually is, but as methods for solving recurrences go, it's pretty mellow.

Unwinding

• An example: Solve \( T(n) = 2T(n/2) + n \)

| What information is missing from this recurrence, which we will need to be able to solve it? |

• *Unwind* the recurrence by plugging in the definition on successively smaller arguments:
  - From the definition, \( T(n) = 2T(n/2) + n \)
  - \( T(n) = 2[2T(n/4) + n/2] + n \)
  - \( = 4T(n/4) + 2n \)
  - \( = 8T(n/8) + 3n \)

Do you see a pattern here? And when does this unwinding end?
Unwinding

• An example: Solve $T(n) = 2T(n/2) + n$
• For a base case, let’s use $T(1) = 1$ (or $\theta(1)$, if we want)

• *Unwind* the recurrence by plugging in the definition on successively smaller arguments:
  – From the definition, $T(n) = 2T(n/2) + n$

\[
\begin{align*}
T(n) &= 2T(n/2) + n \\
&= 2[2T(n/4) + n/2] + n \\
&= 4[2T(n/8) + n/4] + 2n \\
&= 8T(n/16) + 3n \\
&\vdots \\
&= 2^k[T(n/2^k)] + k*n \\
&\vdots \\
&= n^*T(1) + (lg n)*n \\
&= \theta(n \ lg n)
\end{align*}
\]

This name may make it sound more relaxing than it actually is, but as methods for solving recurrences go, it’s pretty mellow.
Recursion-Tree Method

- An example: Mergesort

\[
T(n) = \begin{cases} 
0 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases}
\]

What’s the cost at each tree-level (i.e., not counting levels below it)?

What’s the recursion tree structure?

• Set up a tree to total up the work done by the algorithm

• Tree structure for complexity analysis corresponds to tree of recursive calls by the algorithm

• Total work by the algorithm: Sum of work at all levels of the tree
Recursion-Tree Method

• An example: Mergesort

\[
T(n) = \begin{cases} 
0 & \text{if } n = 1 \\
2T(n/2) & \text{sorting both halves} \\
T(n/4) + n & \text{merging otherwise}
\end{cases}
\]