Merge Sort and Recurrences

COSC 581, Algorithms
January 14, 2014
Reading Assignments

• Today’s class:
  – Chapter 2, 4.0, 4.4

• Reading assignment for next class:
  – Chapter 4.2, 4.5
3 Common Algorithmic Techniques

• Divide and Conquer
• Dynamic Programming
• Greedy Algorithms
Divide and Conquer

• Recursive in structure
  – *Divide* the problem into sub-problems that are similar to the original but smaller in size
  – *Conquer* the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  – *Combine* the solutions to create a solution to the original problem
An Example: Merge Sort

**Sorting Problem:** Sort a sequence of $n$ elements into non-decreasing order.

- **Divide:** Divide the $n$-element sequence to be sorted into two subsequences of $n/2$ elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.
Merge Sort – Example

Original Sequence

Sorted Sequence
Merge-Sort \((A, p, r)\)

**INPUT:** a sequence of \(n\) numbers stored in array \(A\)

**OUTPUT:** an ordered sequence of \(n\) numbers

\[
\text{MergeSort} (A, p, r) \quad \text{// sort } A[p..r] \text{ by divide & conquer}
\]

1. if \(p < r\)
2. \(\text{then } q \leftarrow \lfloor (p+r)/2 \rfloor\)
3. \(\text{MergeSort} (A, p, q)\)
4. \(\text{MergeSort} (A, q+1, r)\)
5. \(\text{Merge} (A, p, q, r) \quad \text{// merges } A[p..q] \text{ with } A[q+1..r]\)

**Initial Call:** \(\text{MergeSort}(A, 1, n)\)
**Procedure Merge**

```pseudo
Merge(A, p, q, r)
1  \( n_1 \leftarrow q - p + 1 \)
2  \( n_2 \leftarrow r - q \)
3    for \( i \leftarrow 1 \) to \( n_1 \)
4      do \( L[i] \leftarrow A[p + i - 1] \)
5    for \( j \leftarrow 1 \) to \( n_2 \)
6      do \( R[j] \leftarrow A[q + j] \)
7  \( L[n_1+1] \leftarrow \infty \)
8  \( R[n_2+1] \leftarrow \infty \)
9  \( i \leftarrow 1 \)
10 \( j \leftarrow 1 \)
11  for \( k \leftarrow p \) to \( r \)
12    do if \( L[i] \leq R[j] \)
13      then \( A[k] \leftarrow L[i] \)
14          \( i \leftarrow i + 1 \)
15    else \( A[k] \leftarrow R[j] \)
16      \( j \leftarrow j + 1 \)
```

**Input:** Array containing sorted subarrays \( A[p..q] \) and \( A[q+1..r] \).

**Output:** Merged sorted subarray in \( A[p..r] \).

**Sentinels**, to avoid having to check if either subarray is fully copied at each step.
Merge – Example

A

... 1 6 8 9 26 32 42 43 ...  

\( k \)

\( L \)

6 8 26 32 \( \infty \)

\( R \)

1 9 42 43 \( \infty \)
Correctness of Merge

**Merge**(\(A, p, q, r\))

1. \(n_1 \leftarrow q - p + 1\)
2. \(n_2 \leftarrow r - q\)
3. for \(i \leftarrow 1\) to \(n_1\) do \(L[i] \leftarrow A[p + i - 1]\)
4. for \(j \leftarrow 1\) to \(n_2\) do \(R[j] \leftarrow A[q + j]\)
5. \(L[n_1 + 1] \leftarrow \infty\)
6. \(R[n_2 + 1] \leftarrow \infty\)
7. \(i \leftarrow 1\)
8. \(j \leftarrow 1\)
9. for \(k \leftarrow p\) to \(r\) do if \(L[i] \leq R[j]\)
10. then \(A[k] \leftarrow L[i]\)
11. i \leftarrow i + 1
12. else \(A[k] \leftarrow R[j]\)
13. j \leftarrow j + 1

**Loop Invariant for the for loop**
At the start of each iteration of the for loop:

Subarray \(A[p..k - 1]\) contains the \(k - p\) smallest elements of \(L\) and \(R\) in sorted order.

\(L[i]\) and \(R[j]\) are the smallest elements of \(L\) and \(R\) that have not been copied back into \(A\).

**Initialization:**
Before the first iteration:

- \(A[p..k - 1]\) is empty.
- \(i = j = 1\).
- \(L[1]\) and \(R[1]\) are the smallest elements of \(L\) and \(R\) not copied to \(A\).
Correctness of Merge

**Merge(A, p, q, r)**

1. \( n_1 \leftarrow q - p + 1 \)
2. \( n_2 \leftarrow r - q \)
3. for \( i \leftarrow 1 \) to \( n_1 \)
   4. do \( L[i] \leftarrow A[p + i - 1] \)
5. for \( j \leftarrow 1 \) to \( n_2 \)
   6. do \( R[j] \leftarrow A[q + j] \)
7. \( L[n_1+1] \leftarrow \infty \)
8. \( R[n_2+1] \leftarrow \infty \)
9. \( i \leftarrow 1 \)
10. \( j \leftarrow 1 \)
11. for \( k \leftarrow p \) to \( r \)
   12. do if \( L[i] \leq R[j] \)
   13. then \( A[k] \leftarrow L[i] \)
   14. \( i \leftarrow i + 1 \)
15. else \( A[k] \leftarrow R[j] \)
16. \( j \leftarrow j + 1 \)

**Maintenance:**

**Case 1: \( L[i] \leq R[j] \)**

- By LI, \( A \) contains \( p - k \) smallest elements of \( L \) and \( R \) in sorted order.
- By LI, \( L[i] \) and \( R[j] \) are the smallest elements of \( L \) and \( R \) not yet copied into \( A \).
- Line 13 results in \( A \) containing \( p - k + 1 \) smallest elements (again in sorted order). Incrementing \( i \) and \( k \) reestablishes the LI for the next iteration.

**Similarly for \( L[i] > R[j] \).**

**Termination:**

- On termination, \( k = r + 1 \).
- By LI, \( A \) contains \( r - p + 1 \) smallest elements of \( L \) and \( R \) in sorted order.
- \( L \) and \( R \) together contain \( r - p + 3 \) elements. All but the two sentinels have been copied back into \( A \).
Analysis of Merge Sort

• Running time $T(n)$ of Merge Sort:
• Divide: computing the middle takes $\Theta(1)$
• Conquer: solving 2 subproblems takes $2T(n/2)$
• Combine: merging $n$ elements takes $\Theta(n)$
• Total:

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

$$\Rightarrow T(n) = \Theta(n \lg n) \text{ (CLRS, Chapter 4)}$$
Recurrences

• Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs
• Often used to define a recursive algorithm’s runtime
• Example: $T(n) = 2T(n/2) + n$
Recurrence Relations

• Equation or an inequality that characterizes a function by its values on smaller inputs.

• **Solution Methods** (Chapter 4)
  – Substitution Method.
  – Recursion-tree Method.
  – Master Method.

• Recurrence relations **arise when we analyze the running time of iterative or recursive algorithms.**
  – **Ex:** Divide and Conquer.
    
    \[ T(n) = \Theta(1) \quad \text{ if } n \leq c \]
    
    \[ T(n) = a \cdot T(n/b) + D(n) + C(n) \quad \text{ otherwise} \]
Substitution Method

- **Guess** the form of the solution, then use **mathematical induction** to show it correct.
  
  – Substitute guessed answer for the function when the inductive hypothesis is applied to smaller values – hence, the name.

- Works well when the solution is easy to guess.

- No general way to guess the correct solution.
Example 1 – Exact Function

Recurrence:

\[ T(n) = 1 \quad \text{if} \quad n = 1 \]
\[ T(n) = 2T(n/2) + n \quad \text{if} \quad n > 1 \]

\textbf{Guess:} \quad T(n) = n \log n + n.

\textbf{Induction:}

- **Basis:** \( n = 1 \Rightarrow n \log n + n = 1 = T(n) \).
- **Hypothesis:** \( T(k) = k \log k + k \) for all \( k < n \).
- **Inductive Step:**
  \[ T(n) = 2T(n/2) + n \]
  \[ = 2 ((n/2)\log(n/2) + (n/2)) + n \]
  \[ = n \log(n/2) + 2n \]
  \[ = n \log n - n + 2n \]
  \[ = n \log n + n \]
Recursion-tree Method

• Making a good guess is sometimes difficult with the substitution method.

• Use recursion trees to devise good guesses.

• Recursion Trees
  – Show successive expansions of recurrences using trees.
  – Keep track of the time spent on the subproblems of a divide and conquer algorithm.
  – Help organize the algebraic bookkeeping necessary to solve a recurrence.
Recursion Tree – Example

• Running time of Merge Sort:

\[ T(n) = \Theta(1) \quad \text{if } n = 1 \]
\[ T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1 \]

• Rewrite the recurrence as

\[ T(n) = c \quad \text{if } n = 1 \]
\[ T(n) = 2T(n/2) + cn \quad \text{if } n > 1 \]

\( c > 0 \): Running time for the base case and time per array element for the divide and combine steps.
Recursion Tree for Merge Sort

For the original problem, we have a cost of $cn$, plus two subproblems each of size $(n/2)$ and running time $T(n/2)$.

Each of the size $n/2$ problems has a cost of $cn/2$ plus two subproblems, each costing $T(n/4)$.

Cost of divide and merge.

Cost of sorting subproblems.
Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1.

\[
\begin{align*}
\text{Total} & : cn\lg n + cn
\end{align*}
\]
Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1.

- Each level has total cost $cn$.
- Each time we go down one level, the number of subproblems doubles, but the cost per subproblem halves $\Rightarrow$ cost per level remains the same.
- There are $\lg n + 1$ levels, height is $\lg n$. (Assuming $n$ is a power of 2.)
- Can be proved by induction.
- Total cost = sum of costs at each level $= (\lg n + 1)cn = cn\lg n + cn = \Theta(n \lg n)$. 
Recursion Trees – Caution Note

• Recursion trees only generate guesses.
  – Verify guesses using substitution method.

• A small amount of “sloppiness” can be tolerated. Why?

• If careful when drawing out a recursion tree and summing the costs, can be used as direct proof.
Summing up Cost of Recursion Trees

• Evaluate:
  – Cost of individual node at depth $i$
  – Number of nodes at depth $i$
  – Total height of tree
Recursion Tree for Merge Sort

Continue expanding until the problem size reduces to 1.

- Cost of node at depth \( i = \frac{cn}{2^i} \)
- Number of nodes at depth \( i = 2^i \)
- Depth of tree
  \( = \) number of times can divide \( cn \) by \( 2^i \)
  until we get value of 1
  \( = \lg n + 1 \)

Putting together:

\[
\sum_{i=0}^{\lg n+1} 2^i \frac{cn}{2^i} = \sum_{i=0}^{\lg n+1} cn
\]

\( = \Theta(n \lg n) \)
Can also write out algebraically...

\[ T(n) = cn + 2T \left( \frac{n}{2} \right) \]

\[ = cn + 2 \left( \frac{cn}{2} + 2T \left( \frac{n}{4} \right) \right) \]

\[ = cn + 2cn/2 + 2 \left( \frac{2cn}{4} + 2T \left( \frac{n}{8} \right) \right) \]

\[ = \ldots \]

\[ = \sum_{i=0}^{\lg n+1} 2^i \frac{cn}{2^i} = \sum_{i=0}^{\lg n+1} cn \]

\[ = \Theta(n \lg n) \]
Example 2

- Formulate (and solve) recursion tree for:
  \[ T(n) = 2T(n - 1) + c \]
Example #3

• Insertion sort can be expressed as a recursive procedure as follows:
  
  – In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.
Example #4

• Argue that the solution to the recurrence:

\[ T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn \]

where \( c \) is constant,

is \( \Omega(n \lg n) \) by appealing to a recursion tree.
Recall 3 Methods for Solving Recurrence Relations

• **Solution Methods** (Chapter 4)
  – Substitution Method -- Today
  – Recursion-tree Method -- Today
  – Master Method -- Next time
Next Time: The Master Method

- Based on the Master theorem.
- “Cookbook” approach for solving recurrences of the form

\[ T(n) = aT(n/b) + f(n) \]

- \( a \geq 1, \ b > 1 \) are constants.
- \( f(n) \) is asymptotically positive.
- \( n/b \) may not be an integer, but we ignore floors and ceilings. Why?

- Requires memorization of three cases.
Reading Assignments

• Reading assignment for next class:
  – Chapter 4.2, 4.5