CS 312: Algorithm Design & Analysis

Lecture #9: Solving Divide and Conquer Recurrence Relations with Change of Variable

Slides by Eric Ringger, with contributions from Mike Jones, Eric Mercer, Sean Warnick

This work is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License.

Announcements

- HW #6 due now
- Questions about Non-homogeneous RR?
- Project #2
  - Whiteboard Experience: Wednesday
  - Early Day: Friday
  - Due: Monday
  - Significantly more time than proj. #1

Objectives

- Understand how to analyze recurrence relations that are typical of divide and conquer algorithms
- Learn to use the “change of variable” technique to solve such recurrences
- Review Mergesort (quickly!)
  - Theoretical analysis using RRs

Divide and Conquer

\[ t(n) = a \cdot t\left(\frac{n}{a}\right) + g(n) \]

General Case:

\[ t_n = a \cdot t_{n/2} + g(n) \]

Example: Binary Search

\[ T(n) = T\left(\frac{n}{2}\right) + 1 \]

Analysis Approach: Reduction

<table>
<thead>
<tr>
<th>Divide and Conquer Recurrence Relation</th>
<th>Change of Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Homogeneous Recurrence Relation with Geometric Forcing Function</td>
<td></td>
</tr>
<tr>
<td>LTII Homogeneous Recurrence Relation</td>
<td></td>
</tr>
</tbody>
</table>

Analysis of Binary Search

Example: \( T(n) = T\left(\frac{n}{2}\right) + 1 \)

Initial cond. \( T(1) = 1 \)

Change of variable: let \( \begin{cases} \end{cases} \)

\[ T(2^k) = T\left(2^{k-1}\right) + 1 \]

One more substep: \( t_k = T(2^k) \)

\[ t_k = t_{k-1} + 1 \]

\( t_0 = t_{k-1} = 1 \)
Binary Search: Continued

Apply Change Eqn. Then:

\[ b_k = b_{k-1} = 1 = b_k \]

Characteristic Func.:

(All): Non-Homogeneous

LHS: \( b_k = b_{k-1} \Rightarrow b_k = b_{k-2} + b_{k-1} \)

Divide and Conquer Recurrence Relation

Non-Homogeneous Recurrence Relation with Geometric Forcing Function

LTI Homogeneous Recurrence Relation

Unwind the Stack

Analysis Approach: Reduction

Review: Merge sort

- Split the array in 1/2
- Sort each 1/2 -- recursively
- Adhoc(): Use insertion sort when sub-arrays are small
- Merge the sorted halves

```
3 1 4 1 5 9 2 6 5 3 5 8 9
3 1 4 1 5 9 2 6 5 3 5 8 9
1 1 3 4 5 9 2 3 5 5 8 9
1 1 2 3 4 5 5 5 6 8 9
```

```
procedure merge(U[p+1], V[q+1], A[1..n])
for k = 1 to p+q do
    if U[p+1] < V[q+1] then
        A[k] = U[p+1]; i = i + 1
    else
        A[k] = V[q+1]; j = j + 1
```

Analysis of Binary Search (cont.)

\[ T(1) = c_1 + c_2 \log_2 1 = c_1 \]

```
9 8 6 5 5 3 2
```

Analysis of merge()

How long to merge?

```
3 1 4 1 5 9 2 6 5 3 5 8 9
3 1 4 1 5 9 2 6 5 5 5 8 9
1 1 3 4 5 9 2 3 5 5 6 8 9
1 1 2 3 4 5 5 5 6 8 9
  A
  U
  V
```
Merge sort

\[ T(n) \]

procedure mergesort (A[1..n])
if \( n \) is small enough then insertsort (A)
else
array U[1..floor(n/2)], V[1..ceil(n/2)] \( O(n) \)
\( U[1..floor(n/2)] \leftarrow A[1..floor(n/2)] \) \( O(n) \)
\( V[1..ceil(n/2)] \leftarrow A[1+floor(n/2)..n] \) \( O(n) \)
mergesort (U) \( T(n/2) \)
mergesort (V) \( T(n/2) \)
merge (U, V, A) \( O(n) \)

What is the efficiency of Mergesort?
\[ T(n) = 2T(n/2) + g(n) \quad g(n) \in \Theta(n) \]

Using the Master Theorem

- \( a = \) number of sub-instances that must be solved
- \( n = \) original instance size (variable)
- \( n/b = \) size of subinstances
- \( d = \) polynomial order of \( g(n) \),
  where \( g(n) = \) cost of dividing and recombing

\[ t(n) = a \cdot t(n/b) + O(n^d) \]

\[ t(n) = \begin{cases} 
O(n^d) & \text{if } a < b^d \\
O(n^d \log n) & \text{if } a = b^d \\
O(n^d \log_a a) & \text{if } a > b^d 
\end{cases} \]

Efficiency of Mergesort

\[ t(n) = \begin{cases} 
O(n^d) & \text{if } a < b^d \\
O(n^d \log n) & \text{if } a = b^d \\
O(n^d \log_a a) & \text{if } a > b^d 
\end{cases} \]

\( \text{Note: } r=2 \ \text{and} \ \text{master theorem} \)

5 solutions:

- \( k = 2^k \)
- \( T(k) \)

General case:

- \( t_k = T(2^k) \)
- \( T(2^k) = c_1 \cdot 2^k + c_2 \cdot 2^k \)

- \( t_k = T(2^k) \)
- \( T(2^k) = c_1 \cdot 2^k + c_2 \cdot 2^k \)

- \( t_k = T(2^k) \)
- \( T(2^k) = c_1 \cdot 2^k + c_2 \cdot 2^k \)

- \( t_k = T(2^k) \)
- \( T(2^k) = c_1 \cdot 2^k + c_2 \cdot 2^k \)

- \( t_k = T(2^k) \)
- \( T(2^k) = c_1 \cdot 2^k + c_2 \cdot 2^k \)
Another Useful Logarithm Identity

\[ a^{\log_a n} = n^{\log_a a} \]

**Proof:**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a^{\log_a n} = (\log_a a)^{\log_a n} )</td>
<td>( x^y ) and ( \log_y y ) are inverses</td>
</tr>
<tr>
<td>( (\log_a a)^{\log_a n} = n^{\log_a a} )</td>
<td>( (x^y)^z = x^{yz} )</td>
</tr>
<tr>
<td>( \log a^{\log a} n = \log_{\log a} n \log a )</td>
<td>Commutativity of multiplication.</td>
</tr>
<tr>
<td>( \log a^{\log a} n = (\log a)^{\log a} n )</td>
<td>( (x^y)^z = x^{yz} )</td>
</tr>
<tr>
<td>( (\log a)^{\log a} n = n^{\log a a} )</td>
<td>( x^y ) and ( \log x, y ) are inverses</td>
</tr>
</tbody>
</table>

**Assignment**

- Read: Section 2.3 in the textbook
- HW #7:
  - Part III Exercises (Section 3.2)
  - Analyze 3-part mergesort using recurrence relations techniques
  - Problem 2.4 in the textbook (using the master theorem where possible)