Complexity Analysis of Recursive Algorithms, Divide and Conquer Algorithms

Rosen Ch. 7.1: Recurrence Relations
Rosen Ch. 7.3: Divide & Conquer
Walls Ch. 10.2: Advanced Sorting Algorithms
Recurrence Relations: An Overview

- What is a recurrence relation?
  - A recursively defined sequence

- Example
  - Arithmetic progression: \( a, a+d, a+2d, \ldots, a+nd \)
    - \( a_0 = a \)
    - \( a_n = a_{n-1} + d \)
Recursively defined functions and recurrence relations

- A recursive function
  - $A(0) = a$ (base case)
  - $A(n) = A(n-1) + d$ for $n > 0$ (recursive part)

The above recursively defined function generates the sequence defined on the previous slide
  - $a_0 = a$
  - $a_n = a_{n-1} + d$

- A recurrence relation produces a sequence, an application of a recursive function produces a value from the sequence
Solving recurrence relations

Solve \( a_0 = 2; \ a_n = 3a_{n-1} + 2 \) for \( n > 0 \)

Solve by substitution

Use formula for summing geometric series
\[ 1 + r + r^2 + \ldots + r^n = \frac{r^{n+1} - 1}{r - 1} \] if \( r \neq 1 \)

Warning: deriving a closed form solution using backward substitutions is not always straightforward
Computation Time for Recursive Algorithms

Example: Compute the factorial function $N!$

```c
int factorial(int N) {
    if n==0 return 1;
    else return factorial(N - 1) * N;
}
```

The number of operations required can be characterized by:

$$f(n) = f(n-1) + 1$$

What series is generated by this recurrence relation?

$$f(n) = f(n-2) + 1 + 1 = f(n-3) + 1 + 1 + 1 = \ldots$$

need an initial condition (corresponds to base case of recursion)

$$f(0) = 1$$
Recursive Algorithms

**Example**: Tower of Hanoi, move all disks to third peg without ever placing a larger disk on a smaller one.
Recursive Algorithms

Example: Tower of Hanoi, move all disks to third peg without ever placing a larger disk on a smaller one.
Recursive Algorithms

**Example:** Tower of Hanoi, move all disks to third peg without ever placing a larger disk on a smaller one.

\[ f(n) = f(n-1) + \ldots \]
Recursive Algorithms

Example: Tower of Hanoi, move all disks to third peg without ever placing a larger disk on a smaller one.

\[ f(n) = f(n-1) + 1 + \ldots \]
Recursive Algorithms

**Example:** Tower of Hanoi, move all disks to third peg without ever placing a larger disk on a smaller one.

\[
f(n) = f(n - 1) + 1 + f(n - 1)\\
\]

\[
f(n) = 2f(n - 1) + 1, \quad f(1) = 1
\]
Recursive Algorithms

Example: Tower of Hanoi, move all disks to third peg without ever placing a larger disk on a smaller one.

\[ f(n) = 2f(n - 1) + 1, \quad f(1) = 1 \]

How to figure out an explicit formula for this sequence:

• Substitution
• Theory of Recurrence Relations
Recurrence Relations: Formal Definition

A recurrence relation for the sequence \( \{a_n\} \) is an equation that expresses \( a_n \) in terms of one of more of the previous terms of the sequence, namely, \( a_0, a_1, \ldots a_{n-1} \), for all integers \( n \) with \( n \geq n_0 \) where \( n_0 \) is a nonnegative integer.

- sequence = recurrence relation + initial conditions ("base case")
- Example: \( a_n = 2a_{n-1} + 1, \ a_1 = 1 \)
You deposit $10,000 in a savings account that yields 10% yearly interest. How much money will you have after 30 years?

\[ b_n = b_{n-1} + rb_{n-1} = (1 + r)^n b_0 \]
Fibonacci’s Rabbits

- Suppose a newly-born pair of rabbits, one male, one female, are put on an island.
  - A pair of rabbits doesn’t breed until 2 months old.
  - Thereafter each pair produces another pair each month
  - Rabbits never die.

- How many pairs will there be after n months?

image from: http://www.jimloy.com/algebra/fibo.htm
Recurrence Examples

Suppose a string of decimal digits is a valid codeword if it contains an even number of 0s. How many such codewords?

- Base case: $a_1 = 9$
- Recurrence:
  - Extend valid string with digit $\neq 0$
  - Append 0 to invalid string of length $n-1$

$$a_n = 9a_{n-1} + (10^{n-1} - a_{n-1}) = 8a_{n-1} + 10^{n-1}$$
Divide-and-Conquer

Basic idea: Take large problem and divide it into smaller problems until problem is trivial, then combine parts to make solution.

Recurrence relation for the number of steps required:

\[ f(n) = a \cdot f(n/b) + g(n) \]

- \( n/b \) - the size of the sub-problems solved
- \( a \) - number of sub-problems
- \( g(n) \) - steps necessary to combine solutions to sub-problems
Example: Binary Search

public static int binSearch (int myArray[], int first, int last, int value) {
    // returns the index of value or -1 if not in the array
    int index;
    if (first > last) { index = -1; }
    else {
        int mno = (first + last)/2;
        if (value == myArray[mno]) { index = mno; }
        else if (value < myArray[mno]) {
            index = binSearch(myArray, first, mno-1, value);
        } else {
            index = binSearch(myArray, mno+1, last, value);
        }
    }
    return index;
}

What are $a$, $b$, and $g(n)$?

$$f(n) = a \cdot f(n/b) + g(n)$$
Example: Finding Max and Min in unsorted array

Algorithm:

- If \( n = 1 \), then element is the max \textit{and} min.
- If \( n > 1 \), divide sequence in half, find max/min of each and choose lowest (min) and highest (max) from each half

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

What are \( a, b, \) and \( g(n) \)?
Estimating big-O (Master Theorem)

Let \( f \) be an increasing function that satisfies

\[
f(n) = a \cdot f(n/b) + c \cdot n^d
\]

whenever \( n = b^k \), where \( k \) is a positive integer, \( a \geq 1 \), \( b \) is an integer \( > 1 \), and \( c \) and \( d \) are real numbers with \( c \) positive and \( d \) nonnegative. Then

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

From section 7.3 in Rosen
Example: Binary Search using the Master Theorem

\[ f(n) = a f(n / b) + n^d \]

\[ f(n) = O(?) \]

\[ f(n) = \begin{cases} 
O(n^d) & \text{if } a < b^d \\
O(n^d \log n) & \text{if } a = b^d \\
O(n^{\log_b a}) & \text{if } a > b^d 
\end{cases} \]
Max and Min in unsorted array

\[ f(n) = a f(n / b) + n^d \]

\[ f(n) = O(?) \]

\[
\begin{align*}
    f(n) &= \begin{cases} 
    O(n^d) & \text{if } a < b^d \\
    O(n^d \log n) & \text{if } a = b^d \\
    O(n^{\log_b a}) & \text{if } a > b^d 
    \end{cases}
\end{align*}
\]
Aside: Sorting Redux from 161

- Simple Sorts: Bubble, Insertion, Selection
- Doubly nested loop
- Outer loop puts one element in its place
- It takes i steps to put element i in place
  - $n-1 + n-2 + n-3 + \ldots + 3 + 2 + 1$
  - $O(n^2)$ complexity
  - In place
Merge Sort

- Basic idea
  - Divide data into two (smaller) parts
  - Sort the smaller parts
  - Merge the sorted parts
  - Divide and conquer!
Merge Sort - Divide

{7,3,2,9,1,6,4,5}

{7,3,2,9} {1,6,4,5}

{7,3} {2,9} {1,6} {4,5}

{7} {3} {2} {9} {1} {6} {4} {5}
Merge Sort - Merge

{1,2,3,4,5,6,7,9}

{2,3,7,9}  {1,4,5,6}

{3,7}  {2,9}  {1,6}  {4,5}

{7}  {3}  {2}  {9}  {1}  {6}  {4}  {5}
public static void mergesort(Comparable[] theArray, int first, int last) {
// Sorts the items in an array into ascending order.
// Precondition: theArray[first..last] is an array.
// Postcondition: theArray[first..last] is sorted.
if (first < last) {
    int mid = (first + last) / 2;  // midpoint of the array
    mergesort(theArray, first, mid);  // mid of the array
    mergesort(theArray, mid + 1, last);
    merge(theArray, first, mid, last);
}  // if first >= last, there is nothing to do
}
Merge sort demo

Merge Sort: Space requirements

- Can divide be done in place?
  - Yes, easy

- Can merge sort be done in-place?
  - Keeping the unmerged parts sorted
  - In O(n) time?
Merge using two arrays

Data:

```
2 3 7 9 1 4 5 6
```

Temp:

Step 1:

```
1
2 3 7 9 1 4 5 6
```

Step 2:

```
1 2
2 3 7 9 1 4 5 6
```

Step 3:

```
1 2 3
2 3 7 9 1 4 5 6
```

Step 4:

```
1 2 3 4
2 3 7 9 1 4 5 6
```
Merge - Using Two Arrays

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 5:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 6:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Step 7:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Step 8:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
mergeSort – Complexity

\[ \{1,2,3,4,5,6,7,9\} \]

\[ \{2,3,7,9\} \quad \{1,4,5,6\} \]

\[ \{3,7\} \quad \{2,9\} \]

\[ \{7\} \quad \{3\} \quad \{2\} \quad \{9\} \]

\[ \{1,6\} \quad \{4,5\} \]

\[ \{1\} \quad \{6\} \quad \{4\} \quad \{5\} \]

- At depth i
  - work done
    - split
    - merge
  - total work?
- Total depth?
- Total work?

\[ O(n \log(n)) \]
mergeSort: Recurrence Analysis

\[ f(n) = a \cdot f\left(\frac{n}{b}\right) + cn^d \]

- \( a = \)
- \( b = \)
- \( c = \)
- \( d = \)
- \( O(?) \)

\[ f(n) = \begin{cases} 
O(n^d) & \text{if } a < b^d \\
O(n^d \log n) & \text{if } a = b^d \\
O(n^{\log_b a}) & \text{if } a > b^d 
\end{cases} \]
Stable Sorting Algorithms

- Suppose we are sorting a database of users according to their name. Users can have identical names.

- A **stable** sorting algorithm maintains the relative order of records with equal keys (i.e., sort key values). Stability: whenever there are two records \( R \) and \( S \) with the same key and \( R \) appears before \( S \) in the original list, \( R \) will appear before \( S \) in the sorted list.

- Is mergeSort stable?
MergeSort – Summary

- Efficient
- Predictable performance
Quick Sort

- Basic idea:
  - Select a pivot element
  - Subdivide array into 3 parts
    - pivot in its sorted position
    - sub-array of elements <= pivot
    - sub-array of elements >= pivot
  - Recursively apply to each sub-array
Quick Sort - Partitioning

P < < > ? ? ? ??
public static void quickSort(Comparable[] theArray, int first, int last) {
    // Sorts the items in an array into ascending order.
    // Precondition: theArray[first..last] is an array.
    // Postcondition: theArray[first..last] is sorted.
    int pivotIndex;
    if (first < last) {  // create the partition: S1, Pivot, S2
        pivotIndex = partition(theArray, first, last);
        // sort regions S1 and S2
        quickSort(theArray, first, pivotIndex-1);
        quickSort(theArray, pivotIndex+1, last);
    }
}

why does it work?
Quick Sort Code

private static int partition(Comparable[] theArray, int first, int last) {
    // Precondition: theArray[first..last] is an array; first <= last.
    // Postcondition: Returns the index of the pivot element of theArray[first..last]. // Upon completion of the method, this will be the index value lastS1 such that // S1 = theArray[first..lastS1-1] < pivot
    // theArray[lastS1] == pivot
    // S2 = theArray[lastS1+1..last] >= pivot
    Comparable tempItem; // holder for swaps
    choosePivot(theArray, first, last); // place pivot in theArray[first]
    Comparable pivot = theArray[first]; // reference pivot
    // initially, everything but pivot is in unknown
    int lastS1 = first; // index of last item in S1
Quick Sort Code

// move one item at a time until unknown region is empty
for (int firstUnknown = first + 1; firstUnknown <= last;
    ++firstUnknown) {
    // Invariant: theArray[first+1..lastS1] < pivot
    // move item from unknown to proper region
    if (theArray[firstUnknown].compareTo(pivot) < 0) {
        // item from unknown belongs in S1
        ++lastS1;
        templItem = theArray[firstUnknown]; theArray[firstUnknown] =
        theArray[lastS1];
        theArray[lastS1] = templItem; }
    // else item from unknown belongs in S2 }
    // place pivot in proper position and mark its location
        templItem = theArray[first]; theArray[first] = theArray[lastS1];
    theArray[lastS1] = templItem;
    return lastS1; }
When things go bad...

- **Worst case**
  - quicksort is $O(n^2)$ when the array is already sorted and the smallest item is chosen as the pivot

<table>
<thead>
<tr>
<th>Original array:</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pivot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Pivot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Pivot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Pivot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>First partition:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pivot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>$S_1$ is empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_1$ is empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_1$ is empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_1$ is empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DEMO
quickSort – Algorithm Complexity

- **Depth of call tree?**
  - $O(\log n)$  *split roughly in half, best case*
  - $O(n)$  *worst case*

- **Work done at each depth**
  - $O(n)$

- **Total Work**
  - $O(n \log n)$  *best case*
  - $O(n^2)$  *worst case*
quickSort: Recurrence Analysis

\[ f(n) = a \cdot f \left( \frac{n}{b} \right) + cn^d \]

- a =
- b =
- c =
- d =
- O(?)
Strategies for Pivot Selection

- First value
  - Worst case - if array is already sorted

- Middle value
  - Better for sorted data, same as previous case for random; worst case can still happen.

- Median of 3 sample values
  - Worst case $O(n^2)$ can still happen
    - but less likely
Improvements

- Recursion incurs overhead
  - Dominates cost for small arrays

- Hybrid sort algorithm
  - Use quicksort for large partitions
  - Use bubble, insertion or selection sort for smaller arrays
Space requirement for quicksort

- How much memory does quicksort require?
How fast can we sort?

- **Observation:** all the sorting algorithms so far are *comparison sorts*
  - A comparison sort must do at least $O(n)$ comparisons (*why?*)
  - We have an algorithm that works in $O(n \log n)$
  - What about the gap between $O(n)$ and $O(n \log n)$

- **Theorem:** all comparison sorts are $\Omega(n \log n)$

- MergeSort is therefore an “optimal” algorithm
Sorting in linear time!

- Counting sort
  - No comparisons between elements.
  - But…depends on assumption about the numbers being sorted:
    - We assume numbers are in the range $1,\ldots,k$
Sort by Counting

1. CountingSort(A, k)
2. for i=1 to k
3. \( C[i] = 0; \)
4. for j=0 to n-1
5. \( C[A[j]] += 1; \)
6. // B array of size n
7. for i=1 to k
8. add i to B \( C[i] \) times

Complexity?

Input: A[1..n], where A[j] ∈ {1, 2, 3, …, k}
Output: B[1..n], sorted
C[1..k] for auxiliary storage
Radix Sort (by MSD)

1. Take the most significant digit (MSD) of each number.
2. Sort the numbers based on that digit, grouping elements with the same digit into one bucket.
3. Recursively sort each bucket, starting with the next digit to the right.
4. Concatenate the buckets together in order.

```
80  24  62  40  68  20  26

24, 20, 26   40     62, 68    80

20  24  26  40       62       68  80
```
Radix Sort

- To avoid using extra space: Radix sort by Least Significant Digit

RadixSort(A, d)

   // d - number of digits
   for i=1 to d
       sort(A) on the i^{th} least significant digit

Assumption: sort(A) is a stable sort

Show Example.

What to do if not all numbers have the same # of digits?
Radix Sort

- **Can we prove it will work?**
- Sketch of an inductive argument (induction on the number of passes):
  - Assume lower-order digits are sorted
  - Show that sorting next digit $i$ leaves array correctly sorted:
    - If two digits at position $i$ are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order
Radix Sort

- Radix sort is:
  - Fast
  - Asymptotically fast (i.e., $O(n)$)
  - Simple to code
  - A good choice

- Can we use it for strings?

- So why not use it for every application?