**Divide and Conquer Algorithms**, Complexity Analysis of Recursive Algorithms

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

---

**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?

$$f(n) = f(n-2) + 1 + 1 = f(n-3) + 1 + 1 + 1 = ...$$

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.

$$f(n) = f(n-1) + ...$$
Recursive Algorithms

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

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

How to figure out an explicit formula for this sequence:
- Substitution
- Theory of Recurrence Relations

Recurrence Relations

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, \( a_{n-1}, a_{n-2}, \ldots, a_{n_0} \), for all integers \( n \geq n_0 \) where \( n_0 \) is a nonnegative integer.

- \( \text{sequence} = \text{recurrence relation} + \text{initial conditions} \) ("base case")
- Example: \( a_n = 2a_{n-1} + 1, \quad a_1 = 1 \)

Compound Interest

You deposit $10,000 in a savings account that yields 10% yearly interest. How much money will you have after 30 years?

\[ b_n = b_{n-1} + rb_{n-1} = (1 + r)b_{n-1} \]

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?

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: \( a = 9 \)
- Recurrence:
  - Extend valid string with digit \( \neq 0 \)
  - Append 0 to invalid string of length \( n-1 \)

\[ a_n = 9a_{n-1} + \left( 10^{n-1} - a_{n-1} \right) = 8a_{n-1} + 10^{n-1} \]
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:

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

- \( n/b \) - 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

```java
public static int binarySearch(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 = binarySearch(myArray, first, mno-1, value); }
        else { index = binarySearch(myArray, mno+1, last, value); }
    }
    return index;
}
```

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

Estimating big-O (Master Theorem)

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

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

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

\[
\begin{align*}
  f(n) & = O(n^d) & \text{if } a < b^d \\
  f(n) & = O(n^d \log n) & \text{if } a = b^d \\
  f(n) & = O(n^{d+\epsilon}) & \text{if } a > b^d
\end{align*}
\]

From section 7.3 in Rosen

Example: Finding Max and Min in unsorted array

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

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

- What are \( a \), \( b \), and \( g(n) \)?

Example: Binary Search using the Master Theorem

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

\[
\begin{align*}
  f(n) & = O(n^d) & \text{if } a < b^d \\
  f(n) & = O(n^d \log n) & \text{if } a = b^d \\
  f(n) & = O(n^{d+\epsilon}) & \text{if } a > b^d
\end{align*}
\]

Example: Max and Min in unsorted array

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

\[
\begin{align*}
  f(n) & = O(n^d) & \text{if } a < b^d \\
  f(n) & = O(n^d \log n) & \text{if } a = b^d \\
  f(n) & = O(n^{d+\epsilon}) & \text{if } a > b^d
\end{align*}
\]
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

Divide-and-Conquer Sorting

- Algorithm Framework:
  - Break the array up in two
  - Sort recursively
  - We look at two strategies
    - Break array in two (almost) equal parts and merge the results: Merge Sort
    - Break array in two parts left and right such that elements in left <= elements in right: Quick Sort

Merge Sort

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

Merge Sort - Divide

- Merge Sort - Merge

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:
Temp:

Step 1:
Step 2:
Step 3:
Step 4:

Merge - Using Two Arrays

Step 5:
Step 6:
Step 7:
Step 8:

MergeSort

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);
        mergesort(theArray, mid + 1, last);
        merge(theArray, first, mid, last);
    } // if first >= last, there is nothing to do
}

mergeSort – Complexity

mergeSort: Recurrence Analysis

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

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 Code
```java
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);
    }
}
```

Quick Sort - Partitioning
```java
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() chooses the pivot
    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
    for (int firstUnknown = first + 1; firstUnknown <= last; ++firstUnknown) {
        // use pivot to partition
        if (theArray[firstUnknown].compareTo(pivot) < 0) {
            // item from unknown belongs in S1
            ++lastS1;
            tempItem = theArray[firstUnknown]; theArray[firstUnknown] = theArray[lastS1];
            theArray[lastS1] = tempItem;            }
        // else item from unknown belongs in S2
    }
    // place pivot in proper position and mark its location
    tempItem = theArray[first];  theArray[first] = theArray[lastS1];
    theArray[lastS1] = tempItem;
    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>Old partition</th>
<th>New partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 2 7 1 8 3</td>
<td>5 2</td>
<td>7 1 8 3</td>
</tr>
<tr>
<td>5 2</td>
<td>1</td>
<td>3 7 8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1 3 7 8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1 3 7 8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1 3 7 8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1 3 7 8</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

$$f(n) = af(n/b) + cn^d$$

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

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**
  - More conservative
  - 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
  - Theorem: all comparison sorts are $\Omega(n \log n)$
    - A comparison sort must do $O(n \log n)$ comparisons (why?)
    - What about the gap between $O(n)$ and $O(n \log n)$
  - 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 $1\ldots k$
    - Input: $A[1..n]$, where $A[i] \in \{1, 2, 3, \ldots, k\}$
    - Output: $B[1..n]$, sorted
    - Also: Array $C[1..k]$ for auxiliary storage

---

Counting Sort

```python
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
```

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

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\textsuperscript{th} least significant digit
```

Show Example.

What happens if not all numbers have the same # of digits?
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 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 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?