Divide-and-Conquer Algorithms

Analysis of Algorithms
Divide and Conquer Paradigm

1. Divide the problem into smaller sub-problems
2. Conquer the smaller sub-problems via recursive calls
3. Combine the solutions of the sub-problems into a solution for the original problem
Mergesort

• Classic divide-and-conquer algorithm
• Central idea: If you have two sorted lists, you can create a combined sorted list if you merge the lists
• We know that the smallest value will be the first one in either of the two smaller lists
• If we move the smallest value to the new list, we can repeat the process until the entire list is sorted
Mergesort Implementation

```java
public class Merge {
    private static void merge(...) {
        /* See next slide */
    }

    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid + 1, hi);
        merge(a, aux, mid + 1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void sort(Comparable[] a) {
        aux = new Comparable[a.length];
        sort(a, aux, 0, a.length - 1);
    }
}
```
Example: Assume we have already copied A, G, H, I, and L to a[], and now we are comparing M (aux[j]) with O (aux[i]). Since M < O, a[k] = aux[j], and j is incremented.
Mergesort Application

• Question: Imagine you have a 20GB file with one string per line. Explain how you would sort this file.
Analysis of Mergesort

• Mergesort divides the list in half each time, so the difference between the best and worst case is how much work merge does.

• Notice that a list of N elements gets broken into two lists of N/2 elements that are recursively sorted and then merged together.

• Thus, the running time is proportional to N log N for the recursion plus N for merging.

• Total = N log N + N \approx N \log N
Analysis of Mergesort

To find the total amount of work that mergesort does, multiply the total number of recursive calls by the amount of work that gets done at each level (the merge operation). For N data items, there are $\log N$ recursive calls, and each call is responsible for merging N items. Thus, the total amount of work = $\log N \times N = N \log N$. 
## Insertion Sort vs. Mergesort

- Empirical analysis based on running time estimates assuming randomly-ordered data and:
  - Laptop executing $10^8$ comparisons per second
  - Supercomputer executing $10^{12}$ comparisons per second

<table>
<thead>
<tr>
<th>Computer</th>
<th>Thousand Items</th>
<th>Million Items</th>
<th>Billion Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>Instant</td>
<td>2.8 hours</td>
<td>317 years</td>
</tr>
<tr>
<td>Super</td>
<td>Instant</td>
<td>1 second</td>
<td>1 week</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Computer</th>
<th>Thousand Items</th>
<th>Million Items</th>
<th>Billion Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>Instant</td>
<td>1 second</td>
<td>18 minutes</td>
</tr>
<tr>
<td>Super</td>
<td>Instant</td>
<td>Instant</td>
<td>Instant</td>
</tr>
</tbody>
</table>

*Moral: You can spend a lot of money and a lot of time, or you can use a good algorithm.*
In-Place Sorting

- Mergesort uses extra space proportional to $N$
- Definition: A sorting algorithm is \textit{in-place} if it uses $\leq constant \times \log N$ extra memory
- Insertion sort, selection sort, Shellsort are considered to be in-place
- Challenge for the bored: develop an in-place merge algorithm for use with mergesort
Comparison-Based Sorting

• Consider a “cost model” that determines the efficiency of a sorting algorithm based on the number of comparisons made
  – You could use a decision tree to count the number of comparisons made

• How many comparisons must be made at most and at least for any comparison-based sorting algorithm?
  – Count sort, bucket sort, and radix sort are not comparison-based sorts, so they do not apply here
Comparison-Based Sorting

Sort \((a, b, c)\), where \(a\), \(b\), and \(c\) represent any 3 distinct integers:

(at least) one leaf for each possible ordering
Comparison-Based Sorting

• Assume an array consists of $N$ distinct values $a_1$ through $a_N$
• To sort this array, the worst case is dictated by height $h$ of the decision tree
• A binary tree of height $h$ has at most $2^h$ leaves
• There are $N!$ possible orderings; therefore we need at least $N!$ leaves

- Question: True or False: Sorting 6 elements with a comparison-based sort requires at least 10 comparisons in the worst case
Comparison-Based Sorting

• Mergesort is optimal with respect to the number of comparisons
  – "Optimal" means that the best case number of comparisons = the worst case number of comparisons
  – Don’t waste your time trying to design a comparison-based sorting algorithm that guarantees $\frac{1}{2} N \log N$ compares

• However, mergesort is not optimal with respect to space usage
Counting Inversions

• Consider the problem of *rankings*
• How can you compare two different rankings for similarity?
  – Netflix uses collaborative filtering (a *recommender system*) to match your movie preferences with those of other Netflix users
  – Google uses *meta-search* tools, where the search engine executes a query using a suite of different optimization algorithms, and then the results are compared for similarity of rankings
Counting Inversions

• One way to compare two rankings is to represent one of the rankings as a list of N (distinct) numbers and define a measure that tells us how far this list is from being in sorted order
  – The measure is 0 if the list is in sorted order (a perfect match to the list 1, 2, 3, ...), and increases as the list become less sorted (less of a match to 1, 2, 3, ...)
  – This measure is the same as the number of pairs that are out of order (i.e., the number of inversions in the list)
Counting Inversions

For example, there are 3 inversions in the list \((2, 4, 1, 3, 5)\):

```
1          2          3          4          5
2          4          1          3          5
```

Each crossing pair of line segments indicates an inversion: \((2,1), (4,1), \) and \((4,3)\)
Counting Inversions

8 inversions – (2,1), (7,1), (7,3), (7,5), (7,4), (5,4), (7,6), (8,6)
Counting Inversions

• Input: an array $A$ containing the numbers 1 to $N$ in an order that represents a ranking
• Output: the number of inversions (i.e., the number of pairs $(i, j)$ of array indices with $i < j$ and $A[i] > A[j]$)

• Example: if $A = [1, 3, 5, 2, 4, 6]$, then output 3 because there are 3 inversions in this list
  – The greatest number of inversions for a list of 6 items is 15 ($(6 \times 5)/2 = 15$); thus, the “goodness of the match” is $3/15 = 0.2$
Counting Inversions

• How to find the number of inversions?
• You could look at every pair of numbers to see if they create an inversion – runtime of $N^2$
• Can we do better using a divide-and-conquer approach? Yes!
• Use a variation of mergesort to so that it runs in $N \log N$ time instead of $N^2$
Counting Inversions

- Basic idea: Classify the kinds of inversions we could possibly have into one of 3 types
- Consider an inversion \((i, j)\) with \(i < j\)
  - Left inversion: both of the indices \(i, j \leq N/2\)
  - Right inversion: both of the indices \(i, j > N/2\)
  - Split index: \(i \leq N/2 < j\)
- Note that we can compute all left and right inversions recursively
- We need a separate routine to compute the split inversions after the recursion
Counting Inversions

countInversions (array A, length n) {
    if (n == 1) return 0;
    else {
        (B, x) = countInversions(left half of A, n/2);
        (C, y) = countInversions(right half of A, n/2);
        (D, z) = countSplits(B, C, n);
        return x + y + z;
    }
}

B is the sorted sub-array returned from the first recursive call
C is the sorted sub-array returned from the second recursive call
D is the final sorted array A returned from countSplits
Goal: implement countSplits in linear (N) time (similar to merge)
Counting Inversions

• Consider the step in mergesort where you have two sorted sub-arrays, B and C, and you are about to merge them together
• Keep track of the number of pairs \((b, c)\) with \(b\) in B and \(c\) in C, and \(c < b\)
• Every time an element from C is appended to the final list, increment \(z\) by the number of elements remaining in B
Counting Inversions

2 7 1 3 5 4 8 6

2 7 1 3

2 7 1 3

2 7 1 3

1 3

5 4 8 6

5 4

5 4

8 6

8 6

1 inversion

1 inversion

3 inversions

3 inversions

1 inversion

1 inversion

3 inversions
Counting Inversions

• Is the countInversions algorithm based on mergesort correct?
• Claim: the split inversions involving an element y of the second array C are precisely the numbers left in the first array B when y is copied to the output D
• Proof: let x be an element of the first array B
  1. If x is copied to D before y, then x < y; therefore there is no inversion involving x and y
  2. If x is copied to D after y, then y < x; therefore x and y are involved in a split inversion, as are all of the remaining elements of B
Matrix Multiplication

N = 2:

\[
\begin{bmatrix}
a & b \\
c & d
\end{bmatrix} \cdot \begin{bmatrix}
e & f \\
g & h
\end{bmatrix} = \begin{bmatrix}
ae + bg & af + bh \\
ce + dg & cf + dh
\end{bmatrix}
\]

Iterative algorithm for matrix multiplication is proportional to \(N^3\). We need to do \(N^3 = 2^3 = 8\) multiplications plus \(N^2 = 4\) additions. Can we do better using a divide-and-conquer technique?
Matrix Multiplication

• Idea #1: Divide the big matrix into sub-matrices each of size N/2 x N/2

• Write $X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ and $Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$ where $A, B, C, D, E, F, G,$ and $H$ are sub-matrices of size N/2 x N/2

• The sub-matrices “behave” mathematically as if they are just atomic elements (integers)

• Thus, $X \cdot Y = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$
Matrix Multiplication

- Recursive algorithm #1:
  - Step 1: Compute 8 sub-products recursively
    - $\chi \cdot \gamma = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$
  - Step 2: Do the necessary additions
- Overall runtime is still proportional to $N^3$
- Can we do better?
Matrix Multiplication

• Idea #2: Strassen’s algorithm (1969):
  – Step 1: Compute only 7 sub-products recursively, instead of 8 sub-products recursively
  – Step 2: Do the necessary (clever) additions and subtractions

• Overall runtime is proportional to $N^{2.81}$
Matrix Multiplication

The 7 sub-products are:  
\[ P_1 = A(F-H), \quad P_2 = (A+B)H, \quad P_3 = (C+D)E, \]
\[ P_4 = D(G-E), \quad P_5 = (A+D)(E+H), \quad P_6 = (B-D)(G+H), \quad P_7 = (A-C)(E+F) \]

Claim:
\[
\begin{bmatrix}
AE + BG & AF + BH \\
CE + DG & CF + DH
\end{bmatrix} = 
\begin{bmatrix}
P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\
P_3 + P_4 & P_1 + P_5 - P_3 - P_7
\end{bmatrix}
\]

Proof: 
\[
AE + BG = AE + AH + DE + DH + DG - DE - AH - BH + BG + BH - DG - DH = AE + BG
\]

Do the same for the other 3 summations.

QED!
Basic Quicksort algorithm:

1. Partition the array so that, for some j
   - Entry $a[j]$ is in place
   - Smaller entries are to the left of j
   - Larger entries are to the right of j

2. Sort left half and right half recursively
Quicksort

Partition: repeat until i and j pointers cross

- Scan i from left to right so long as \((a[i] < a[lo])\)
- Scan j from right to left so long as \((a[j] > a[lo])\)
- Exchange \(a[i]\) with \(a[j]\)

Stop i here.

Stop j here and exchange \(a[i]\) with \(a[j]\) (R and C).
Quicksort

• When pointers cross, exchange $a[lo]$ with $a[j]$
• This completes one pass of the partition
• Recursively partition the left half and the right half
Quicksort Trace

Initial values
random shuffle

no partition
for subarrays
of size 1

result

Quicksort trace (array contents after each partition)
Quicksort Implementation

private static int partition(Comparable[] a, int lo, int hi) {
    int i = lo, j = hi+1;
    while (true) {
        while (less(a[++i], a[lo]))
            if (i == hi) break;

        while (less(a[lo], a[--j]))
            if (j == lo) break;

        if (i >= j) break;
        exch(a, i, j);
    }

    exch(a, lo, j);
    return j;
}
Quicksort Implementation

```java
public class Quick {
    private static int partition(Comparable[] a, int lo, int hi) {
        /* see previous slide */
    }

    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }
}

shuffle needed for performance guarantee (stay tuned)
```
Analysis of Quicksort

• Correctness of quicksort: proof by induction
• Let P(n) be the statement “quicksort correctly sorts every input of length n”
• Prove P(n) for all n ≥ 1 by induction:
  1. (Base case): P(1) holds because an input of length 1 is correctly sorted
  2. (Inductive step): For every n ≥ 2, if P(k) holds for all k < n, then P(n) holds as well
     • See next slide for proof of inductive step
Analysis of Quicksort

• For the inductive step, recall that quicksort partitions the array around some pivot $p$, and at the end of each partition, the pivot winds up in the correct position.

<table>
<thead>
<tr>
<th>&lt; $p$</th>
<th>$p$</th>
<th>&gt; $p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>left half</td>
<td>right half</td>
<td></td>
</tr>
</tbody>
</table>

• Let $k_1$ be the length of the left half and $k_2$ be the length of the right half ($k_1$ and $k_2$ must be less than $n$, why?)

• By the inductive hypothesis then, both the left half and the right half gets sorted correctly by the recursive calls $P(k_1)$ and $P(k_2)$.
Analysis of Quicksort

• Best case runtime
  – The number of compares is $\sim N \log N$

• Worst case runtime
  – The number of compares is $\sim \frac{1}{2} N^2$

• Average case runtime
  – The number of compares is $\sim 1.39 N \ln N$
  – The number of exchanges is $1/3 N \log N$
Analysis of Quicksort

• The runtime of quicksort depends crucially on the choice of a good pivot
  – Ideally, we’d like the pivot to partition the array into two equally-sized sub-arrays for the recursion
  – A pivot that is the median value of the array will partition the array into equal halves

• How to choose a good pivot?
  – In every recursive call, choose the pivot randomly
  – A random pivot is “good enough often enough” to achieve N log N runtime in the average case

• True or False: An adversary can force randomized quicksort to run in quadratic time by providing an already sorted array, or a reverse-sorted array, of size N
Mergesort vs. Quicksort

- Empirical analysis based on running time estimates assuming randomly-ordered data and:
  - Laptop executing $10^8$ comparisons per second
  - Supercomputer executing $10^{12}$ comparisons per second

<table>
<thead>
<tr>
<th>Computer</th>
<th>Thousand Items</th>
<th>Million Items</th>
<th>Billion Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>Instant</td>
<td>1 second</td>
<td>18 minutes</td>
</tr>
<tr>
<td>Super</td>
<td>Instant</td>
<td>Instant</td>
<td>Instant</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Computer</th>
<th>Thousand Items</th>
<th>Million Items</th>
<th>Billion Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>Instant</td>
<td>0.6 seconds</td>
<td>12 minutes</td>
</tr>
<tr>
<td>Super</td>
<td>Instant</td>
<td>Instant</td>
<td>Instant</td>
</tr>
</tbody>
</table>
Sorting versus Selection

• Selection:
  – Used to find order statistics
  – Example: find the $k$ smallest items in an array

• Goal: given an array of $N$ items, find the $k^{th}$ smallest
  – Could sort the array and look at the $k^{th}$ item
    • For example, min ($k=0$), max ($k=n-1$), median ($k=n/2$)
    • Using sorting to solve selection is an example of a reduction
  – Can we do better? What about linear runtime?
Sorting versus Selection

• Is selection as hard as sorting or is it easier?
• Selection is **fundamentally easier** than sorting
• Idea: consider quicksort, but only look at one of the sub-arrays after each partition
  – Quick-select (k-select) algorithm takes linear time on average, but quadratic time in the worst case
  – Each partition splits the array approximately in half: $N + N/2 + N/4 + ... + 1 \approx 2N$ compares
Question: Suppose we are looking for the 5\textsuperscript{th} order statistic (5\textsuperscript{th} smallest) in an input array of length 10. Assume after an initial partition, that the pivot ends up in the third position of the partitioned array. Also assume that the input array begins at index 1 (not 0). On which side of the pivot do we recurse, and what order statistic should we look for?

a) The 3\textsuperscript{rd} order statistic on the left side of the pivot
b) The 2\textsuperscript{nd} order statistic on the right side of the pivot
c) The 5\textsuperscript{th} order statistic on the right side of the pivot
d) Not enough information to answer the question – we might need to recurse on the left or the right side of the pivot
Sorting versus Selection

\[\text{kSelect}(A, n, k)\]
if \(n == 1\) return \(A[1]\)
Choose pivot \(p\) from \(A\) uniformly at random
Partition \(A\) around \(p\) and let \(j = \text{new index of } p\)
if \(j == k\) return \(p\)
if \(j > k\) return \(\text{kSelect}(\text{left half of } A, j-1, k)\)
if \(j < k\) return \(\text{kSelect}(\text{right half of } A, n-j, k-j)\)
Choosing a Sorting Algorithm

• Sorting applications have diverse needs:
  – Stable?
  – Parallel?
  – Deterministic?
  – Keys all distinct?
  – Multiple key types?
  – Linked list or arrays?
  – Large or small items?
  – Is your array randomly ordered?
  – Need guaranteed performance?

• An elementary sort may be the method of choice for some combination of needs
# Comparison-Based Sorting Summary

<table>
<thead>
<tr>
<th>Sort</th>
<th>In-place?</th>
<th>Stable?</th>
<th>Worst</th>
<th>Best</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>Yes</td>
<td>No</td>
<td>$\frac{n^2}{2}$</td>
<td>$\frac{n^2}{2}$</td>
<td>$\frac{n^2}{2}$</td>
</tr>
<tr>
<td>Insertion</td>
<td>Yes</td>
<td>Yes</td>
<td>$\frac{n^2}{2}$</td>
<td>$n$</td>
<td>$\frac{n^2}{4}$</td>
</tr>
<tr>
<td>Shell</td>
<td>Yes</td>
<td>No</td>
<td>?</td>
<td>$n$</td>
<td>?</td>
</tr>
<tr>
<td>Quick</td>
<td>Yes</td>
<td>No</td>
<td>$\frac{n^2}{2}$</td>
<td>$n \log n$</td>
<td>$2n \ln n$</td>
</tr>
<tr>
<td>3-way Quick</td>
<td>Yes</td>
<td>No</td>
<td>$\frac{n^2}{2}$</td>
<td>$n$</td>
<td>$2n \ln n$</td>
</tr>
<tr>
<td>Merge</td>
<td>No</td>
<td>Yes</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>Heap</td>
<td>Yes</td>
<td>No</td>
<td>$2n \log n$</td>
<td>$n \log n$</td>
<td>$2n \log n$</td>
</tr>
<tr>
<td>Holy Grail</td>
<td>Yes</td>
<td>Yes</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
</tr>
</tbody>
</table>