**Sorting**

Lecture 11  
CS2110 – Summer 2009

---

### BubbleSort

- Simplest sorting algorithm
- Works well when input is nearly sorted
- Worst-case is $O(n^2)$
- Consider reverse-sorted input
- Best-case is $O(n)$
- Expected-case is $O(n^2)$

```java
//sort a[], an array of int
boolean done; int n = a.length;
do {
done = true; --n;
for (int i = 1; i <= n; ++i)
   if (a[i] < a[i - 1]) {
temp = a[i]; a[i] = a[i - 1]; a[i - 1] = temp;
done = false;
}
} while (!done);
```

---

### InsertionSort

- Many people sort cards this way
- Invariant: everything to left of \( i \) is already sorted
- Works especially well when input is nearly sorted
- Worst-case is $O(n^2)$
- Consider reverse-sorted input
- Best-case is $O(n)$
- Expected case is $O(n^2)$

```java
//sort a[], an array of int
for (int i = 1; i < a.length; i++) {
temp = a[i];
k = i;
for (; 0 < k && temp < a[k - 1]; k--)
a[k] = a[k - 1];
a[k] = temp;
}
```

---

### SelectionSort

- This is the other common way for people to sort cards
- Worst-case $O(n^2)$
- Consider reverse-sorted input
- Best-case $O(n^2)$
- Expected-case $O(n^2)$

```java
//sort a[], an array of int
for (int i = 0; i < a.length - 1; i++) {
int min = i;
for (int j = i + 1; j < a.length; j++)
   if (a[j] < a[min]) min = j;

int temp = a[i]; a[i] = a[min]; a[min] = temp;
}
```

---

### MergeSort

- Quintessential divide-and-conquer algorithm
- Divide array into equal parts, sort each part, then merge
- Runtime
  - Worst-case $O(n \log n)$
  - Best-case $O(n \log n)$
  - Expected-case $O(n \log n)$

```java
//sort a[], an array of int
for (int i = 0; i < a.length; i++) {
for (int j = i + 1; j < a.length; j++)
   if (a[i] > a[j]) {
temp = a[i]; a[i] = a[j]; a[j] = temp;
}
}
```

---

### Divide & Conquer?

- It often pays to
  - Break the problem into smaller subproblems,
  - Solve the subproblems separately, and then
  - Assemble a final solution
- This technique is called divide-and-conquer
  - Caveat: It won't help unless the partitioning and assembly processes are inexpensive
- Can we apply this approach to sorting?
Merging Sorted Arrays $A$ and $B$

- Create an array $C$ of size $= \text{size of } A + \text{size of } B$
- Keep three indices:
  - $i$ into $A$
  - $j$ into $B$
  - $k$ into $C$
- Initialize all three indices to 0 (start of each array)
- Compare element $A[i]$ with $B[j]$, and move the smaller element into $C[k]$
- Increment $i$ or $j$, whichever one we took, and $k$
- When either $A$ or $B$ becomes empty, copy remaining elements from the other array ($B$ or $A$, respectively) into $C$

$C =$ merged array

MergeSort Analysis

- Outline (detailed code on the website)
  - Split array into two halves
  - Recursively sort each half
  - Merge the two halves
- Merge = combine two sorted arrays to make a single sorted array
  - Rule: always choose the smallest item
  - Time: $O(n)$ where $n$ is the combined size of the two arrays
- Runtime recurrence
  - Let $T(n)$ be the time to sort an array of size $n$
  - $T(n) = 2T(n/2) + O(n)$
  - $T(1) = 1$
- Can show by induction that $T(n)$ is $O(n \log n)$
- Alternately, can see that $T(n)$ is $O(n \log n)$ by looking at tree of recursive calls

QuickSort

- Intuitive idea
  - Given an array $A$ to sort, choose a pivot value $p$
  - Partition $A$ into two subarrays, $AX$ and $AY$
    - $AX$ contains only elements $\leq p$
    - $AY$ contains only elements $\geq p$
  - Sort subarrays $AX$ and $AY$ separately
  - Concatenate (not merge!) sorted $AX$ and $AY$ to get sorted $A$
  - Concatenation is easier than merging -- $O(1)$

MergeSort Notes

- Asymptotic complexity: $O(n \log n)$
  - Much faster than $O(n^2)$
- Disadvantage
  - Need extra storage for temporary arrays
  - In practice, this can be a disadvantage, even though MergeSort is asymptotically optimal for sorting
  - Can do MergeSort in place, but this is very tricky (and it slows down the algorithm significantly)
- Are there good sorting algorithms that do not use so much extra storage?
  - Yes: QuickSort
QuickSort Questions

- **Key problems**
  - How should we choose a pivot?
  - How do we partition an array in place?

- **Partitioning in place**
  - Can be done in $O(n)$ time (next slide)

- Choosing a pivot
  - Ideal pivot is the median, since this splits array in half
  - Computing the median of an unsorted array is $O(n)$, but algorithm is quite complicated
  - Popular heuristics:
    - Use first value in array (usually not a good choice)
    - Use middle value in array
    - Use median of first, last, and middle values in array
    - Choose a random element

In-Place Partitioning

- How can we move all the blues to the left of all the reds?
  - Keep two indices, LEFT and RIGHT
  - Initialize LEFT at start of array and RIGHT at end of array
  - Invariant: all elements to left of LEFT are blue
    - all elements to right of RIGHT are red
  - Keep advancing indices until they pass, maintaining invariant

QuickSort Analysis

- **Runtime analysis (worst-case)**
  - Partition can work badly, producing this: $P \geq P$
  - Runtime recurrence
    - $T(n) = T(n-1) + n$
  - This can be solved to show worst-case $T(n)$ is $O(n^2)$

- **Runtime analysis (expected-case)**
  - More complex recurrence
  - Can solve to show expected $T(n)$ is $O(n \log n)$

- **Improve constant factor by avoiding QuickSort on small sets**
  - Switch to InsertionSort (for example) for sets of size, say, $\leq 9$
  - Definition of small depends on language, machine, etc.

Sorting Algorithm Summary

- **The ones we have discussed**
  - BubbleSort
  - InsertionSort
  - SelectionSort
  - MergeSort
  - QuickSort

- **Other sorting algorithms**
  - ShellSort (will revisit this)
  - RadixSort
  - BinSort
  - CountingSort

- **Why so many?** Do computer scientists have some kind of sorting fetish or what?
  - Stable sorts: Ins, Sel, Mer
  - Worst-case $O(n \log n)$: Mer, Hea
  - Expected $O(n \log n)$: Mer, Hea, Qui
  - Best for nearly-sorted sets: Ins
  - No extra space needed: Ins, Sel, Hea

- Fastest in practice: Qui
  - Least data movement: Sel
Lower Bound for Comparison Sorting

- Goal: Determine the minimum time required to sort n items
- Note: we want worst-case, not best-case time
  - Best-case doesn’t tell us much; for example, we know Insertion Sort takes O(n) time on already-sorted input
  - Want to know the worst-case time for the best possible algorithm
- But how can we prove anything about the best possible algorithm?
  - We want to find characteristics that are common to all sorting algorithms
  - Let’s limit attention to comparison-based algorithms and try to count number of comparisons

Comparison Trees

- Comparison-based algorithms make decisions based on comparison of data elements
- This gives a comparison tree
  - If the algorithm fails to terminate for some input, then the comparison tree is infinite
  - The height of the comparison tree represents the worst-case number of comparisons for that algorithm
  - Can show that any correct comparison-based algorithm must make at least \( n \log n \) comparisons in the worst case

Java.lang.Comparable<T> Interface

- public int compareTo(T x);
  - Returns a negative, zero, or positive value
    - negative if this is before x
    - 0 if this equals x
    - positive if this is after x
- Many classes implement Comparable
  - String, Double, Integer, Character, Date...
  - If a class implements Comparable, then its compareTo method is considered to define that class’s natural ordering
  - Comparison-based sorting methods should work with Comparable for maximum generality

Lower Bound for Comparison Sorting

- Say we have a correct comparison-based algorithm
  - Suppose we want to sort the elements in an array B[]
  - Assume the elements of B[] are distinct
  - Any permutation of the elements is initially possible
  - When done, B[] is sorted
  - But the algorithm could not have taken the same path in the comparison tree on different input permutations

Lower Bound for Comparison Sorting

- How many input permutations are possible? \( n! \approx 2^{n \log n} \)
  - For a comparison-based sorting algorithm to be correct, it must have at least that many leaves in its comparison tree
  - It must have at least \( n! \approx 2^{n \log n} \) leaves, it must have height at least \( n \log n \) (since it is only binary branching, the number of nodes at most doubles at every depth)
  - Therefore its longest path must be of length at least \( n \log n \), and that it its worst-case running time