CSE 373: Sorting

Chapter 7

Introduction to Sorting

Sorting: One of the most fundamental algorithms

Input: An array $A[]$ of values and its size, $n$.

Output: The array stored in sorted order:
- if $i < j$ then $A[i] \leq A[j], \forall i,j \leq n$

Goals: sort as quickly as possible
- ideally, use $O(1)$ memory (other than $A[]$)
- handle pre-sorted lists quickly
### Insertion Sort

Insertion Sort: One of the simplest sorting algorithms, based on List ADT `Insert()`.

- \( n-1 \) passes
- after pass \( i \), elements \( 0..i \) will be in sorted order
- in pass \( i \), we ripple the \( i^{th} \) element down the array until it’s sorted (with respect to elements \( 0..i-1 \))

### Insertion Sort Example

```
position: 0  1  2  3  4  5
input:    7  4  9  5  8  2
pass 1:
pass 2:
pass 3:
pass 4:
pass 5:
```
**Insertion Sort Analysis**

- Why ripple down rather than up?
- Best case input? Running time?
- Worst case input? Running time?

**Adjacent Swap Algorithms**

A class of algorithms that sort simply by comparing and swapping adjacent elements
- Insertion Sort
- Bubble Sort
- Selection Sort
Inversions

- Given $A[]$, an inversion is a pair $(i,j)$ such that $i < j$, but $A[i] > A[j]$.
  - How many inversions in our example?
    - 7 4 9 5 8 2

- The number of inversions in $A[]$ equals the number of adjacent swaps required to sort it
  - Why?

Average Case Analysis

Q: What is the average number of inversions in a random input array?

A: Consider an arbitrary list $L$ with $n$ unique values
  - Consider the reversal of the list $L_R$
  - Every pair $(i,j)$ represents an inversion in $L$ or in $L_R$
  - The total number of distinct $(i,j)$ pairs is $n(n-1)/2$
  - On average, half of these will be in $L$, half will be in $L_R$
  - Thus, the average array has $n(n-1)/4$ inversions
  - So, adjacent swap algorithms run in $\Theta(n^2)$ on average
Heapsort

- Naive algorithm:
  - Run `BuildHeap()` on the input array
  - Call `DeleteMin()` \( n \) times, storing the results in an output array

- Running Time?

- Disadvantage?

- How can we fix this?
Improved Heapsort

- Use the heap’s array to store the sorted values
- Recall: a $k$-element heap uses the first $k$ positions of its implementing array
- Thus, whenever we delete an element from the heap, store it at the end of the array
- What does this give us?
- How to fix it?

Treesort?

- BSTs can obviously be used to sort input
  - **Insert** all values
  - traverse tree in-order, copying to output array
- This is rarely done in practice (unless a tree is already being used to store the data)
  - asymptotically similar to Heapsort
  - **but** trees require more memory
  - **and** can’t be done using only input array memory
  - might as well use Heapsort