COMP 2140 - Data Structures

Shahin Kamali

Topic 5 - Sorting
University of Manitoba

Based on notes by S. Durocher.
Overview

- Review: Insertion Sort
- Merge Sort
- Quicksort
- Heapsort
- Counting Sort

For further reading, refer to Open Data Structures Book (Chapter 11)
Sorting

**Input:**
- a sequence of \( n \) objects: \( A[0], \ldots, A[n-1] \)
  (typically an array or a linked list)
- a comparison predicate, \( \leq \), that defines a total order on \( A \)

**Output:**
- an ordered representation of the objects in \( A \)

Many sorting algorithms exist:
bubble sort, insertion sort, merge sort, heapsort, radix sort, bucket sort, quicksort, etc.
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i - 1]$ is sorted
  - Insert $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

| 4 | 3 | 2 | 10 | 12 | 1 | 5 | 6 |

| 4 | 3 | 2 | 10 | 12 | 1 | 5 | 6 |
Insertion Sort

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i-1;

        /* Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
Insertion Sort Summary

- **One Iteration of the Insertion Sort Algorithm:**
  - After the $i$th iteration, $A[0..i]$ is sorted.
  - Insert item $A[i+1]$ in its proper place in $A[0..i]$.
  - In the worst case, $i$ items are moved in the $i + 1$th iteration!

![Diagram showing the iteration of Insertion Sort]

- sorted

```
|---|---------|---------|-------|
```

```
|---|---------|-------|---------|
```

- sorted
Insertion Sort Analysis

- In the worst case the array is sorted backwards.

\[
\begin{array}{*{7}{c}}
\hline
n & n-1 & n-2 & \ldots & 3 & 2 & 1 \\
\hline
\vdots
|
\hline
3 & 4 & 5 & \ldots & n & 2 & 1 \\
\hline
2 & 3 & 4 & \ldots & n-1 & n & 1 \\
\hline
1 & 2 & 3 & \ldots & n-2 & n-1 & n \\
\hline
\end{array}
\]

- The total number of moved items:

\[
1 + 2 + \ldots + n - 1 = \frac{n(n - 1)}{2} \in (n^2)
\]
Insertion Sort Time Complexity

- The **worst-case** running time of insertion sort is $O(n^2)$.
- As it turns out, the **average-case** running time is also $O(n^2)$.
- Faster sorting algorithms exist. These include:

<table>
<thead>
<tr>
<th></th>
<th>worst case</th>
<th>average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge Sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heapsort</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>worst case</th>
<th>average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>$O(n^2)$ (random pivot)</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

- The lower bound on the worst-case time complexity of any comparison-based sorting algorithm is $n \log n$. 
Merge Sort

- Merge sort is an example of a *divide-and-conquer* algorithm.
  - Divide the input into two or more disjoint subsets.
  - Recursively solve each sub-problem.
  - Combine solutions to the sub-problems to give the solution to the original problem.
Merge Sort Algorithm

- **Input:** an array $A[0..n − 1]$ of comparable elements.
  - **Divide** $A$ into two subarrays $A[0..\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1, n − 1]$
  - **Recursively** sort each sub-array
  - **Combine** the two subarray via merging them

- The base of recursion is an array of size 1 which is sorted
  - In practice, when the length of sub-array is less than 100, selection sort is applied.
Merge Sort Scheme
Merge Sort Example

MS(A[0..7])

13  -4  7  5  6  9  2  1

Call
return
Merging Sorted Sub-arrays

- Given two sorted arrays $A$ and $B$ of size $n$ and $m$, merge them into array $C$ of size $m + n$.
  - $i$, $j$, and $k$ are three indices moving on $A$, $B$, and $C$.
  - They are initially 0.
- At each step, copy the smaller of $A[i]$ and $B[j]$ to $C[k]$.
  - Increment $k$
  - If $A[i]$ is copied, increment $i$; otherwise increment $j$.
- If one array ends ($i = n$ or $j = m$), copy the remaining items of the other array to $C$. 
Merge Example

length $n$  
A \[\begin{array}{ccccccc}
-5 & -1 & 10 & 21 & 25 & 30 \\
i
\end{array}\]

length $m$  
B \[\begin{array}{cccc}
-3 & -2 & 0 & 15 & 50 \\
j
\end{array}\]

length $n + m$  
C \[\begin{array}{ccccccccc}
k
\end{array}\]

length $n$  
A \[\begin{array}{ccccccc}
-5 & -1 & 10 & 21 & 25 & 30 \\
i
\end{array}\]

length $m$  
B \[\begin{array}{cccc}
-3 & -2 & 0 & 15 & 50 \\
j
\end{array}\]

length $n + m$  
C \[\begin{array}{ccccccccc}
k
\end{array}\]
Java Code for Merging

// Merge A[0..n] and B[0..m] into C[0..n+m-1]
void mergeArrays(int arr1[], int arr2[], int n, int m, int arr3[])
{
    int i = 0, j = 0, k = 0;
    while (i < n && j < m)
    {
        if (A[i] < B[j])
        {
            C[k] = A[i]; i++; k++;
        }
        else
        {
            C[k] = B[j]; j++; k++;
        }
    }
    while (i < n)
    {
        C[k] = A[i]; i++; k++;
    }
    while (j < m)
    {
        C[k] = B[j]; j++; k++;
    }
}
Merge Sort Summary

- Recursively sort the left half of the input array $A$
- Recursively sort the left half of the input array $B$
- Merge the two sub-arrays into a new one
  - note that merging requires a new array, that is, it cannot be done in place.
Time complexity of merge sort

What is the time complexity of merge-sort?

\[ T(n) = \begin{cases} 
1, & n \leq 1 \\
2T(n/2) + O(n), & n \geq 2 
\end{cases} \]

We solve this using replacement method to get \( T(n) \in O(n \log n) \).
Analysis of Merge Sort

\[ T(n) = \begin{cases} 
1, & n \leq 1 \\
2T(n/2) + O(n) & n \geq 2 
\end{cases} \]

Since merging two subarrays of size \( n/2 \) takes \( O(n) \), the merge time is at most \( cn \) for some constant value of \( c \) (think of \( c \) as \( M \) in the definition of big-Oh). So we have:

\[ T(n) \leq 2T(n/2) + c \cdot n \]
\[ \leq 2(2T(n/4) + c \cdot n/2) + c \cdot n = 4T(n/4) + 2cn \]
\[ \leq 4(2T(n/8) + c \cdot n/4) + 2c \cdot n = 8T(n/8) + 3cn \]
\[ \leq \ldots \]
\[ \leq 2^k T\left(\frac{n}{2^k}\right) + kcn \]
\[ \ldots \]
\[ \leq 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + \log ncn = nT(1) + cn \log n \in O(n \log n). \]
Should we use merge sort?

- Unlike insertion sort, merge sort does not work in place.
- Merging two arrays requires temporary storage.
Quick Sort

- Like insertion sort, quicksort works in place.
- Like merge sort, quicksort employs a divide and conquer strategy.
- Quicksort is usually implemented as a randomized algorithm.
Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Any element in the array can be selected as the pivot.
- Typically, the pivot is selected randomly.
- Elements are partitioned into those less than or equal to the pivot and those greater than the pivot.
- In general, elements within a partition are not initially sorted.
- **Partitioning can be performed in place.**
Partition Algorithm

- First, swap pivot with the first element.
- Store elements ≤ the pivot at the front of the array and elements ≥ the pivot at the back of the array.
  1. Scan the array starting from the front until an element is found that is > the pivot.
  2. Scan the array starting from the back until an element is found that is < the pivot.
  3. Swap these two items.
  4. Continue until the entire array has been partitioned.
public static int partition(int[] A, int lo, int hi, int pivot) {
    int finalPivot;
    int left = lo + 1;
    int right = hi;
    swap(A, lo, pivot);
    while (left < right) {
        while (left < right && A[lo] >= A[left])
            left++;
        while (left < right && A[lo] <= A[right])
            right--;
        swap(A, left, right);
    }
    if (A[right] <= A[lo])
        finalPivot = right;
    else
        finalPivot = right - 1;
    swap(A, lo, finalPivot);
    return finalPivot;
}
Quicksort Example

- Sort the following array using quicksort.

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

- **Step 1.** Select a pivot element and swap it with the first element.

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

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

- Note that you could select any element as the pivot.
Quicksort Example

- We now partition the array such that elements to the left of the pivot are \( \leq \) than the pivot and element to the right of the pivot are \( \geq \) the pivot.
  - Initialize left to the leftmost element after the pivot and right to the rightmost element.

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

**left**  
**right**

- **step 2:**  
  Increment left until we find an element that is greater than the pivot.

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

**left**  
**right**
Step 3: Decrement right until we find an element that is less than the pivot.

Step 4: Swap the elements at positions left and right.
Quicksort Example

- **Step 5:** Repeat until $\text{left} \geq \text{right}$.

- **Step 6:** If the element at position $\text{right}$ is less than the pivot, then swap it with the pivot. Otherwise, swap the element at position $\text{right} - 1$ with the pivot.
Quicksort Example

- The pivot element is now in the correct position in the array.
- Elements to the left of the pivot are \( \leq \) than it and elements to the right of the pivot are \( \geq \) than it.

\[
\begin{array}{cccccccc}
1 & -3 & -4 & 2 & 6 & 5 & 3 & 7 \\
\end{array}
\]

- Quicksort is called recursively on the left and right partitions.

\[
\begin{array}{cccccccc}
-4 & -3 & 1 & 2 & 3 & 5 & 6 & 7 \\
\end{array}
\]

- Once the left subarray is recursively sorted and the right subarray is recursively sorted, the entire array is sorted.
- The base case is reached when the array has size \( \leq 1 \).
Java Code for Partition (Review)

// Quicksort Partition Java Code

public static int partition(int[] A, int lo, int hi, int pivot)
{
    int finalPivot;
    int left = lo + 1;
    int right = hi;
    swap(A, lo, pivot);
    swap(A, lo, pivot);
    while (left < right) {
        while (left < right && A[lo] >= A[left])
            left++;
        while (left < right && A[lo] <= A[right])
            right--;
        swap(A, left, right);
    }
    if (A[right] <= A[lo])
        finalPivot = right;
    else
        finalPivot = right - 1;
    swap(A, lo, finalPivot);
    return finalPivot;
}
Java Code for Quick-Sort

```java
public static void quickSort(int [] A) {
    quickSortAux(A, 0, A.length - 1);
}

private static void quickSortAux(int [] A, int lo, int hi){
    if (lo < hi) {
        int pivot = lo +
            (int) (Math.random() * (hi - lo + 1));
        // pivot is selected as a random index in [lo,hi)
        pivot = partition(A, lo, hi, pivot);
        // now pivot is in its right position
        quickSortAux(A, lo, pivot - 1);
        quickSortAux(A, pivot + 1, hi);
    }
}
```
Time Complexity of Quick-Sort

- Recall that we can consider the time complexity in the **worst-case**, **best-case**, and **average-case**.

- We express the time complexity using big-Oh notation!

- What are these time complexities for quick-sort?
Worst-case Analysis of Quick-sort

- In the worst case, the pivot partitions an array of size $n$ into one array of size $n - 1$ and a second array of size 0.
  - This happens when the pivot is the smallest or largest element.
- This leads to the following running time:

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
T(n - 1) + O(n) & \text{if } n \geq 2 
\end{cases}
\]

- For some constant $c$ we have:

\[
T(n) \leq T(n - 1) + cn \\
\leq (T(n - 2) + c(n - 1)) + cn \leq (T(n - 3) + c(n - 2)) + c(n - 1) + cn \\
\leq \ldots \leq T(n - k) + c(n - k + 1) + c(n - k + 2) + \ldots + c(n - 1) + cn \\
\leq \ldots \leq T(1) + c(1) + c(2) + \ldots + c(n - 1) + cn = O(n^2)
\]

(after $k = n$ steps, we get to $T(1)$, which is 1.)
Best-case Analysis of Quick-sort

- In the best case, the pivot partitions an array of size \( n \) into two array of size roughly \( n/2 \).
  - This happens when the pivot is the **median**

- This leads to the following running time:

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + O(n) & \text{if } n \geq 2 
\end{cases}
\]

- For some constant \( c \) we have:

\[
T(n) \leq 2T(n/2) + cn \\
\leq 2(2T(n/4) + cn/2) + cn = 4T(n/4) + 2cn \\
\leq 4(2T(n/8) + c(n/4)) + 2cn = 8T(n/8) + 3cn \\
\leq \ldots \leq 2^k T(n/2^k) + ckn \\
\leq \ldots \leq 2^{\lceil \log n \rceil} T(1) + c\lceil \log n \rceil n = O(n \log n)
\]

(after \( k = \lceil \log n \rceil \) steps, we get to \( T(1) \), which is 1.)
Average-case Analysis of Quick-sort

- As it turns out, the average case running time of quick-sort is similar to the best case.
- This is because making a bad random choice for a pivot is unlikely. Making a bad random choice at every level of the recursion tree is even less likely.
  - This extreme bad case happens when the list is already sorted (decreasing or increasing).
- Furthermore, there exist $O(n)$-time algorithms for finding the median of a set of $n$ values. Using such a median algorithm guarantees an optimal choice of pivot.
Final Notes on Quick-sort

- The running time remains $O(n \log n)$ even if the pivot partitions the array into any fixed fraction.

- For example, say the choice of pivot always guarantees a split of $n = 16$ and $15n/16$. The running time becomes:

  $$T(n) = \begin{cases} 
  1 & \text{if } n \leq 1 \\
  T(n/16) + T(15n/16) + O(n) & \text{if } n \geq 2 
  \end{cases}$$

- For some constant $c$ we have:

  $$T(n) \leq 2T(15n/16) + cn$$

  (after $k = \log_{16/15} n = O(\log n)$ steps, we get to $T(1)$, which is 1.)
Heap-Sort

- **Heap-sort** is another sorting algorithm that is often as efficient as quick-sort.
- It is **in-place** and runs in $O(n \log n)$.
- It is **consistent** in the sense that it performs equally well in the best, average and worst cases.
- Heapsort is implemented using algorithms related to heaps: **insert**, **extractMax**, and **buildHeap**.
Heaps

- It is a **complete binary** tree, i.e., the binary tree is ‘filled’ from top to bottom and from left to right.
- Each node stores a value which is larger than the value of its children.
  - So, the max element is the root!
Heap Operations

- As we will see later, a heap can support the two operations **insert** (add a new element to the tree) and **extract-max** (return and delete the maximum element) efficiently (each in $O(\log n)$ time).

- These operations are associated with a **priority queue** abstract data type.

- **HeapSort**: sorting with a Priority Queue:
  - Insert all $n$ elements into the priority queue in $O(n \log n)$ time.
  - Call extractMax $n$ times to retrieve the elements in reverse order.
public static void priorityQueueSort(int[] A) {
    PriorityQueue pq = new PriorityQueue();
    for (int i = 0; i < A.length; i++)
        pq.insert(A[i]);
    for (int i = A.length - 1; i >= 0; i--)
        A[i] = pq.extractMax();
}

Note, this implementation does not sort in place:
- A new heap is generated and items are copied in it.
Heap Sort In Place

- Step I: Rearrange the input array to form a heap (in place).
  - Simply run `buildHeap` on the array.
- Step II: call `extractMax n` times, storing each result in the next available empty cell in the array.
  - Elements in the heap are stored at the front of the array. Those already sorted are stored at the back of the array.

![Diagram of heap sort in place]
Heap Sort Time Complexity

- We will see later how we can implement a heap so that:
  - Building a heap of \( n \) items take \( O(n) \) time
  - Extract max takes \( O(\log n) \) time
- Step I: re-arranging the array to form a heap takes \( O(n) \) time.
- Step II: calling extract-max \( n \) times takes \( O(n \log n) \) time.

Theorem

*Heap-sort can be implemented to run in \( O(n \log n) \) time in the worst case.*
Heap Sort Summary

- Heap-sort can be implemented in-place, i.e., it does not need to create a new data structure (just re-arrange the input array).
- It is more consistent than the naive implementation of quick-sort, i.e., its worst-case and best-case performance is the same.
- It is slightly slower than the quick-sort with random pivot; but it is considered the most direct competitor of quicksort!
- It is not a divide-and-conquer algorithm → does not need stack memory!
In **comparison-based sorting**, we have a set of objects (e.g., a bag of potato) and an operator that can tell us whether an object is smaller than the other (e.g., a scale for comparing weights of potatoes).

- No other assumption is made for the objects (e.g., they are not necessarily numbers).
- Algorithms like Bubble-sort, Quick-sort, Merge-sort, and Heap-sort are all comparison-based.

Does a comparison-base sorting with time asymptotically less than $O(n \log n)$ exist?

- We show the answer is No!
Lower Bound for Comparison-Based Sorting

- Consider a set of \( n \) distinct objects \( a_1, \ldots, a_n \).
- Any permutation of these objects forms an ordering (a possible input array).
- How many ways they can be ordered?
  - \( n! \) ways, that is, there are \( n! \) possible inputs!
- Sorting corresponds to identifying the permutation of a sequence of elements. Once the permutation is known, the position of each item can be restored.

```
\begin{array}{cccccccc}
2 & 4 & 3 & 8 & 6 & 1 & 5 & 7 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
```
Decision Tree for Sorting an Array

- Suppose an array of $A$ of three elements $[a, b, c]$ is to be sorted.
- Any algorithm can be described with a decision tree for determining the correct sorted order (i.e., the array’s permutation).
  - The number of comparisons made by the algorithm equals to the height of the tree!
  - The number of leaves is $n!$
- Here is the decision tree for one algorithm:
Lower Bound using Decision Tree

- The minimum height of a binary tree with \( n \) nodes is at least \( \log n \).
- Intuitively, whenever two elements are compared (is \( x < y \)?) this eliminates a number of possible permutations. In the worst case, at most half of the remaining possible permutations are eliminated.
- If there are \( n! \) permutations and half are eliminated on each step, it takes at least \( \log_2(n!) \) comparisons to identify the correct permutation of the items.
- So, any algorithm has to make at least \( \log n! \) comparisons.

\[
\log_2(n!) = \log_2(n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1) \\
> \log_2(n \cdot (n-1) \cdot (n-2) \cdots (n/2 + 1) \cdot (n/2)) \\
> \log_2((n/2) \cdot (n/2) \cdot (n/2) \cdots (n/2) \cdot (n/2)) \\
\underbrace{n/2 \text{ times}} \\
= \log_2 \left( (n/2)^{n/2} \right) \\
= \frac{n}{2} \log_2(n/2) \in \Omega(n \log n)
\]
Lower Bound for Comparison-Based Sorting

Theorem

*Any comparison-based sorting has time complexity of $\Omega(n \log n)$*

- We use $\Omega(n \log n)$ to denote the time complexity that is asymptotically at least $n \log n$
Non-comparison based algorithms

- The lower bound of $\Omega(n \log n)$ applies to algorithms that determine a sorted ordering by comparing pairs of elements.
- If the set of elements to be sorted has specific characteristics, then faster sorting algorithms may be possible.
- Such algorithms must use techniques other than comparison alone to determine the sorted order.
- Examples: counting sort, radix sort, bucket sort
Counting Sort

- Let $A$ be an array of $n$ integers in the range $\{0, 1, \ldots, k\}$.
- For each value $i$ in $\{0, 1, \ldots, k\}$, count the number of occurrences of $i$ in $\{0, 1, \ldots, k\}$ and store it in $C[i]$.
- Overwrite $A[0..n - 1]$ with the number of occurrences of each value $\{0, 1, \ldots, k\}$ in sorted order.
Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.
Insertion Sort

```java
//sorting integers in the range [0..k] and stored in A
public static void countingSort(int[] A, int k) {
    int[] C = new int[k + 1];
    for (int i = 0; i <= k; i++)
        C[i] = 0;
    for (int j = 0; j < A.length; j++)
        C[A[j]]++;  
    int index = 0;
    for (int i = 0; i <= k; i++)
        for (int p = 0; p < C[i]; p++)
            A[index++] = i;
}

Unlike comparison based sortings, there are no comparisons between elements of A (all less-than operators test for-loop exit conditions).
```
Counting Sort Running Time

- Counting Sort runs in $O(n + k)$ times.
  - Initializing elements of $C$ to 0 takes $O(k)$ times.
  - Looping over $A$ and setting values of $C$ takes $O(n)$ times.
  - Looping over $C$ and adding sorted elements back to array $A$ takes $O(n + k)$ times.

- Are these worst-case, average-case, or best-case times?
  - No! the code contains no branches (if statements).

- If $k \in O(n \log n)$, then counting sort is at least as efficient as a comparison-based sorting algorithm in the worst case.

- If $k \notin O(n \log n)$, then counting sort is slower than an efficient comparison-based sorting algorithm in the worst case.
Augmented Counting Sort

- Suppose the value of $k$ is initially unknown.
- We can compute $k$:
  
  ```java
  int k = 0;
  for (int j = 0 ; j < A.length ; j++)
      if (A[j] > k) k = A[j];
  ```
- It takes $O(n)$ time
- If $k$ is too large, call an efficient comparison-based sorting algorithm (e.g., quick-sort), otherwise continue with counting sort.
- What if array $A$ contains both positive and negative values?
  - Add a fixed value to all elements to make them non-negative; apply counting sort, and after sorting deduce the added element.
We have examined a few sorting algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>worst case</th>
<th>average case</th>
<th>in place</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>yes</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n^2)$ [naive]</td>
<td>$O(n \log n)$</td>
<td>yes</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>no</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>yes</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>$O(n+k)$</td>
<td>$O(n+k)$</td>
<td>no</td>
</tr>
</tbody>
</table>