Inf 2B: Heapsort and Quicksort

Kyriakos Kalorkoti
School of Informatics
University of Edinburgh

Review of insertionSort and mergeSort

**insertionSort**
- worst-case running time: $\Theta(n^2)$
- sorts in place, is stable

**mergeSort**
- worst-case running time: $\Theta(n \lg(n))$
- does not sort in place (we need the “scratch array” $B$).
  (There is an in-place variation if the input elements are stored in a list.)

---

**maxSort**

**Algorithm** maxSort($A$)
1. for $j \leftarrow A.length - 1$ downto 1 do
2.   $m \leftarrow 0$
3.   for $i = 1$ to $j$ do
4.     if $A[i].key > A[m].key$ then $m \leftarrow i$
5.   exchange $A[m], A[j]$

- heapSort uses the same idea, but it uses a heap to find efficiently the maximum at each step.

---

**Heaps**

Provide efficient access to item with maximum key

**Methods**
- removeMax(): Return and remove an item with max key. [$\Theta(\lg(n))]$
- buildHeap($A$): Turn array $A$ into a heap. [$\Theta(n)$]

Both of these methods build on
- heapify(): Repair heap whose top cell is out of place. [$\Theta(\lg(n))$]
heapSort

Algorithm heapSort(A)
1. buildHeap(A)
2. for j ← A.length – 1 downto 1 do
3. A[j] ← removeMax()

Note: In the above we have implicitly a variable giving the current size of the heap: it starts as n then n – 1 etc.
- buildheap(A) has Θ(n) running-time (n = A.length).
- removeMax() has running-time Θ(lg(j)), if the item removed is j-th lowest in the sorted order.
- The running time of heapSort(A) is:

\[ \Theta(n) + \sum_{j=2}^{n} \Theta(lg j) = \Theta(n \lg n). \]

quickSort

Divide-and-Conquer algorithm:
1. If the input array has strictly less than two elements, do nothing.
   Otherwise, call partition: Pick a pivot key and use it to divide the array into two:
   \[ \leq \text{pivot} \quad \geq \text{pivot} \]
2. Sort the two subarrays recursively.
Algorithm quickSort(A, i, j)
1. if i < j then
2. split ← partition(A, i, j)
3. quickSort(A, i, split)
4. quickSort(A, split + 1, j)

Returned value split satisfies: i ≤ split ≤ j − 1.

Algorithm partition(A, i, j)
1. pivot ← A[i].key
2. p ← i − 1
3. q ← j + 1
4. while TRUE do
5. do q ← q − 1 while A[q].key > pivot
6. do p ← p + 1 while A[p].key < pivot
7. if p < q then
8. exchange A[p], A[q]
9. else return q

After each iteration of the main loop in lines 4–8, for all indices r
- If q ≤ r ≤ j then A[r].key ≥ pivot.
- If i ≤ r < q then A[r].key ≤ pivot.
- Returned value of q satisfies: i ≤ q ≤ j − 1.

Running Time of quickSort

\[
T_{\text{partition}}(n) = \Theta(n)
\]

\[
T_{\text{quickSort}}(n) = \max_{1 \leq s \leq n-1} \left( T_{\text{quickSort}}(s) + T_{\text{quickSort}}(n-s) \right)
+ T_{\text{partition}}(n) + \Theta(1)
= \max_{1 \leq s \leq n-1} \left( T_{\text{quickSort}}(s) + T_{\text{quickSort}}(n-s) \right)
+ \Theta(n).
\]

Implies
\[
T_{\text{quickSort}}(n) = \Theta(n^2)
\]
quickSort (continued)

- quickSort turns out to be very fast in practice.
- Average case running time of quickSort is $\Theta(n \lg(n))$.
- But performs badly ($\Theta(n^2)$) on sorted and almost sorted arrays.

Improvements

- Different choice of pivot (key of middle item, random)
- Use insertionSort for small arrays, etc.

Warning: If you need a sorting algorithm with $\Theta(n \lg n)$ worst case running time then quickSort is NEVER the correct choice! (Unless you enjoy getting 0 for that part of an exercise or exam question.)

```java
public static void quickSort(Item[] A, int i, int j) {
    if (i < j) {
        int split = partition(A, i, j);
        quickSort(A, i, split-1);
        quickSort(A, split+1, j);
    }
}
```

```java
private static int partition(Item[] A, int i, int j) {
    Item tmp;
    Comparable pivot = A[i].key;
    int p = i-1;
    int q = j+1;
    while (true) {
        do q--; while (A[q].key.compareTo(pivot) > 0);
        do p++; while (A[p].key.compareTo(pivot) < 0);
        if (p < q) {
            tmp = A[p];
            A[q] = tmp;
        } else
            return q;
    }
}
```