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)

```
for j ← A.length − 1 downto 1 do
    m ← 0
    for i = 1 to j do
        if A[i].key > A[m].key then m ← i
    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** $j \leftarrow A.length - 1$ **downto** 1 **do**
3. $\quad A[j] \leftarrow$ 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 $\Theta(n)$ running-time ($n = A.length$).
- removeMax() has running-time $\Theta(\lg(j))$, if the item removed is j-th lowest in the sorted order.
- The running time of heapSort(A) is:

$$\Theta(n) + \sum_{j=2}^{n} \Theta(\lg\, j) = \Theta(n \lg\, n).$$

```
private static void heapify(Item[] A,int size,int v) {
    int s;
    if (2*v+1<size && A[2*v+1].key.compareTo(A[v].key)>0)
        s=2*v+1;
    else
        s=v;
    if (2*v+2<size && A[2*v+2].key.compareTo(A[s].key)>0)
        s=2*v+2;
    if (s!=v) {
        Item tmp=A[v];
        A[v]=A[s];
        A[s]=tmp;
        heapify(A,size,s);
    }
}
```

```
public static void heapSort(Item[] A) {
    buildHeap(A,A.length);
    for(int i=A.length; i>=2; i--)
        A[i-1]=removeMax(A,i);
}

private static void buildHeap(Item[] A, int size) {
    for (int v=(size-2)/2; v >= 0; v--)
        heapify(A,size,v);
}

private static Item removeMax(Item[] A,int size) {
    Item i=A[0];
    A[0]=A[size-1];

    heapify(A,size-1,0);
    return i;
}
```

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:

$\leq$ pivot	$\geq$ pivot

2. Sort the two subarrays recursively.

quickSort (continued)

Algorithm quickSort(A, i, j)

1. **if** $i < j$ **then**
2. $\quad$ $split \leftarrow$ partition(A, i, j)
3. $\quad$ quickSort$(A, i, split)$
4. $\quad$ quickSort$(A, split + 1, j)$

Returned value *split* satisfies: $i \leq split \leq j - 1$.

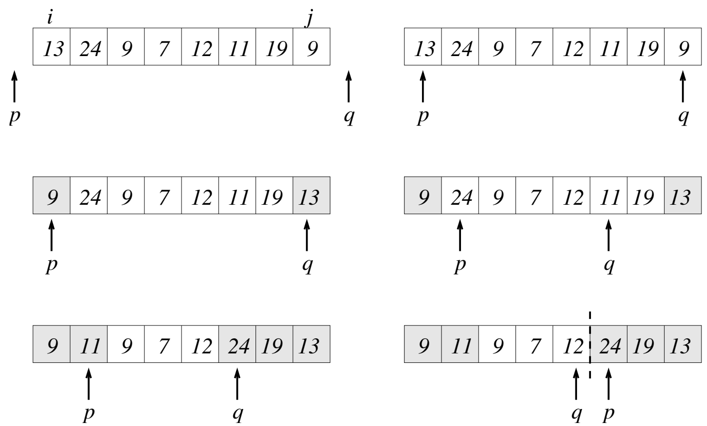
partition

Algorithm partition(A, i, j)

1. $pivot \leftarrow A[i].key$
2. $p \leftarrow i - 1$
3. $q \leftarrow j + 1$
4. **while** TRUE **do**
5. $\quad$ **do** $q \leftarrow q - 1$ **while** $A[q].key > pivot$
6. $\quad$ **do** $p \leftarrow p + 1$ **while** $A[p].key < pivot$
7. $\quad$ **if** $p < q$ **then**
8. $\quad\quad$ exchange $A[p]$, $A[q]$
9. $\quad$ **else return** q

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

partition (continued)



Running Time of quickSort

partition

$$T_{\text{partition}}(n) = \Theta(n)$$

quickSort

$$\begin{aligned} T_{\text{quickSort}}(n) &= \max_{1 \leq s \leq n-1} \big(T_{\text{quickSort}}(s) + T_{\text{quickSort}}(n - s) \big) \\ &\quad + T_{\text{partition}}(n) + \Theta(1) \\ &= \max_{1 \leq s \leq n-1} \big(T_{\text{quickSort}}(s) + T_{\text{quickSort}}(n - s) \big) \\ &\quad + \Theta(n). \end{aligned}$$

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.

```
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);
    }
}
```

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.)

```
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[p]=A[q];
            A[q]=tmp;
        } else
            return q;
    }
}
```