Tutorial 2:
Heapsort, Quicksort, Counting Sort, Radix Sort

Mayank Saksena
September 20, 2006

1 Heapsort

We review Heapsort, and prove some loop invariants for it. For further information, see Chapter 6 of Introduction to Algorithms.

**HEAPSORT**(A)
1 **BUILD-MAX-HEAP**(A)
2 for i = length(A) downto 2
3 swap A[1] and A[i]
4 heap-size(A) = heap-size(A) - 1
5 **MAX-HEAPIFY**(A, 1)

**BUILD-MAX-HEAP**(A)
1 heap-size(A) = length(A)
2 for i = ⌊length(A)/2⌋ downto 1
3 **MAX-HEAPIFY**(A, i)

**MAX-HEAPIFY**(A, i)
1 l = LEFT(i)
2 r = RIGHT(i)
3 if \( l \leq \text{heap-size}(A) \) and \( A(l) > A(i) \) then largest = l
4 else largest = i
5 if \( r \leq \text{heap-size}(A) \) and \( A(r) > A(\text{largest}) \) then largest = r
6 if largest ≠ i
7 swap A[i] and A[largest]
8 **MAX-HEAPIFY**(A, largest)
Loop invariants

First, assume that MAX-HEAPIFY(A, i) is correct, i.e., that it makes the subtree with root i a max-heap. Under this assumption, we prove that BUILD-MAX-HEAP(A) is correct, i.e., that it makes A a max-heap.

We show: at the start of iteration i of the for-loop of BUILD-MAX-HEAP(A) (line 2), each of the nodes i + 1, i + 2, . . . , n is the root of a max-heap.

Initialization Initially, i = [n/2]. Each node [n/2], [n/2] + 1, . . . , n is a leaf, and therefore the root of a max-heap (of size 1).

Maintenance Just before iteration i, by the loop invariant, the children of node i are both roots of max-heaps; then MAX-HEAPIFY(A, i) makes i the root of a max-heap. Thus, also every node j > i is the root of a max-heap. Hence, the loop invariant is established after i has been decremented.

Termination When i = 0, all nodes j > 0 are roots of a max-heap; including node 1.

Now we argue that Heapsort is correct by proving the following loop invariant: At the start of each iteration of the for-loop (line 2), the subarray A[1..i] is a max-heap containing the i smallest elements of A, and the subarray A[i+1..n] contains the n − i largest elements of A sorted.

Initialization Initially, i = n. By line 1, A[1..n] is a max-heap containing the n smallest (all) elements of A, and the subarray A[n+1..n] is empty.

Maintenance Just before some iteration, A[1..i] is a max-heap containing the i smallest elements of A, and the subarray A[i+1..n] contains the n − i largest elements of A sorted, by the loop invariant.

Swapping A[1] and A[i] preserves the order of elements A[i..n], and the elements A[1..i−1] are all smaller than A[i..n].

After decrementing heapsize, the call to MAX-HEAPIFY(A, 1) makes A[1..i−1] a max-heap, restoring the loop invariant before next iteration.

Termination When i = 1, A[1..1] is a max-heap containing the least element of A, and the subarray A[2..n] contains the n − 1 largest elements of A sorted. In other words, A is sorted.

2 Quicksort

QUICKSORT(A, p, r)
1 if p < r
2 q = PARTITION(A, p, r)
3 QUICKSORT(A, p, q − 1)
4 QUICKSORT(A, q + 1, r)
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6      swap A[i] and A[j]
7  swap A[i + 1] and A[r]
8  return i + 1

PARTITION(A, p, r) rearranges A and outputs an index q such that A[p..q - 1] ≤ A[q] < A[q + 1..r]. The following property is a loop invariant (proof left as an exercise) for any index k:
1. If p ≤ k ≤ i, then A[k] ≤ x
2. If i + 1 ≤ k ≤ j - 1, then A[k] > x
3. If k = r, then A[k] = x

Performance analysis
The choice of pivot element (x in PARTITION(A, p, r)) has a major impact on the performance of Quicksort. We analyze the effect of partitioning informally.

The worst possible partitioning would be if we always chose the largest (or smallest) element as pivot – then we get one recursive call of size n - 1 and one of size 0. Since the partitioning is Θ(n), we then get the behaviour: T(n) = T(n - 1) + Θ(n). Expanding, we get T(n) = Θ(n) + Θ(n - 1) + ⋯ + Θ(1) = Θ \left( \frac{n(n+1)}{2} \right) = Θ(n^2).

The best partitioning occurs when at each recursive call, the subproblems are halved, or: T(n) = 2T(n/2) + Θ(n). The Master theorem (a = 2, b = 2, \log_b a = 1) gives T(n) = Θ(n \log n).

Now consider a case when the partitioning always produces a proportional split, e.g.: T(n) = T(\alpha n) + T((1 - \alpha)n) + Θ(n), for some constant 0.5 < \alpha < 1.

We can obtain a bound by e.g. drawing a recursion tree for T(n). Since \alpha > (1 - \alpha), the longest path from the root to a leaf is n \rightarrow \alpha n \rightarrow ⋯ \rightarrow \alpha^k n = 1. Hence the height of tree is \log_{1/\alpha} n. For each level there is a Θ(n) term. Additionally, if the tree was complete, it would have 2^{\log_{1/\alpha} n} = n^{\log_{1/\alpha} 2} leaves each with constant cost. All in all we get T(n) = O(n \log n). As an exercise, prove this using the substitution method.

For further information about Quicksort, see Chapter 7 of Introduction to Algorithms.
### 3 Counting sort

Counting sort sorts integers in \([0, k]\) in time \(\Theta(n + k)\). If \(k = O(n)\), it is therefore \(\Theta(n)\). Array \(B\) is for output, and \(C[0..k]\) for storage.

#### Counting-Sort \((A, B, k)\)

1. for \(i = 0\) to \(k\) \(\quad C[i] = 0\)
2. for \(i = 1\) to \(\text{length}(A)\)
   3. \(C[A(i)] = C[A(i)] + 1\)
3. for \(i = 1\) to \(k\)
   4. \(C[i] = C[i] + C[k - 1]\)
5. for \(i = \text{length}(A)\) down to 1
6. \(B[C[A[i]]] = A[i]\)
7. \(C[A[i]] = C[A[i]] - 1\)

### 4 Radix sort

Radix sort assumes that each element of \(A\) has \(d\) digits, numbered 1 to \(d\), where 1 is the lowest-order digit.

#### Radix-Sort \((A, d)\)

1. for \(i = 1\) to \(d\)
2. use a stable sort to sort \(A\) on digit \(i\)

**Definition** A sorting algorithm is stable if elements with the same value appear in the same order in the output as in the input.

Counting sort is stable. Using counting sort, radix sort is \(\Theta(d(n + k))\), when sorting \(n\) \(d\)-digit numbers where each digit can take \(k\) possible values.

### 5 Recommended problems

Recommended problems from Introduction to Algorithms. I go through some...

- Heapsort time bounds: exercise 6.4-3 on p. 136
- Prove that \(T(n) = T(\alpha n) + T((1 - \alpha)n) + \Theta(n) = O(n \log n)\), where \(0.5 < \alpha < 1\), using e.g. the substitution method
- Quicksort: exercises 7.1-\{1, 2, 3\} p. 148, 7.2-\{1, 3, 4, 5\} p. 153, 7.4-2 p. 159
- Linear time sorting: exercises 8.2-\{2, 3, 4\} p. 170, 8.3-4 p. 173, problem 8-3 p. 179