Test I Study Guide

Illya Starikov

February 15, 2016

1 Sorting Algorithms

1.1 Insertion-Sort

```
Insertion-Sort(A)
    for j = 2 to A.length
        key = A[j]
        // Insert A[j] into the sorted sequence A[1..j - 1]
        i = j - 1
        while i > 0 and A[j] > key
            A[i + 1] =  A[i]
            i = i - 1
        A[i + 1] = key
```

1.2 Merge Sort

```
Merge-Sort(A, p, r)
    if p < r
        q = ⌊(q + r) /2⌋
        Merge-Sort(A, p, q)
        Merge-Sort(A, q + 1, r)
        Merge(A, p, q, r)

Merge(A, p, q, r)
    n_1 = q - p + 1
    n_2 = r - q
    let L[1...n_1 + 1] and R[1..n_2] be new arrays
    for i = 1 to n_1
        L[i] = A[p + i - 1]
    for j = 1 to n_2
        R[j] = A[q + j]
    R[n_1 + 1] = ∞
    R[n_2 + 1] = ∞
    i = 1
    j = 1
```

```
    for k = p to r
        if L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
        else A[k] = R[j]
            j = j + 1
```

1.3 Heap Sort

```
Max-Heapify(A, i)
    l = Left(i)
    r = Right(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        Max-Heaphify(A-largest)

Heapsort(A)
    Build-Max-Heap(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size - 1
        Max-Heapify(A, 1)
```

1.4 Quicksort

```
Quicksort(A, p, r)
    if p < r
        q = Partition(A, p, r)
        Quicksort(A, p, q - 1)
        Quicksort(A, q + 1, r)

Partition(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r - 1
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

1.5 Counting Sort

Let A = input array, B = output array, n = size of array, k = maximum number.

```
countingSort(A, B, n, k)
    let C[0...k] be the counting array
    for i = 0 to k
        C[i] = 0
    for j = 1 to n
        C[A[j]] = C[A[j]] + 1
    for i = 1 to k
        C[i] = C[i] + C[i - 1]
    for j = n down to 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

1.6 Rate of Growth

Insertion Sort Worst: $\Theta(n^2)$, Average: $\Theta(n^2)$

Merge Sort Worst: $\Theta(n \lg n)$, Average: $\Theta(n \lg n)$

Quick Sort Worst: $\Theta(n^2)$, Average: $\Theta(n \lg n)$

Heap Sort Worst: $\Theta(n \lg n)$, Average: $\Theta(n \lg n)$

2 Growth Classes

2.1 O-notation

$O(g(n)) = \{f(n) :$ there exists positive constant c and n_0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0\}$

2.2 Ω-notation

$\Omega(g(n)) = \{f(n) :$ there exists positive constants c and n_0 such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0\}$

2.3 Θ-notation

$$\begin{aligned} \Theta(g(n)) = \{f(n) :\ &\text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ &0 \leq c_1\ g(n) \leq f(n) \leq c_2\ g(n) \text{ for all } n \geq n_0\} \end{aligned}$$

2.4 o-notation (Little-o)

$f(n) \in o(g(n))$ iff for every $c > 0$ there is an $n_0 > 0$ such that

$$0 \leq f(n) < c\ g(n)$$

for all $n \geq n_0$

2.5 ω-notation (Little-omega)

$$f(n) \in \omega(g(n)) \text{ iff } g(n) \in o(f(n)).$$

or $f(n) \in \omega(g(n))$ iff for every $c > 0$ there is an $n_0 > 0$ such that

$$0 \leq c\ g(n) < f(n)$$

for all $n \geq n_0$

3 Master Theorem

The master theorem can only be applied to recurrence equations of the form:

$$T(n) = aT(n/b) + f(n)$$

3.1 Constants

n The size of the problem

a The number of subproblems

n/b The size of each subproblem

f(n) cost outside of recursive calls (divide, combine)

3.2 Cases

$f(n) \in O(n^{\log_b a - \epsilon})$	$T(n) \in \Theta(n^{\log_b a})$
$f(n) \in \Theta(n^{\log_b a})$	$T(n) \in \Theta(n^{\log_b a} \lg n)$
$f(n) \in \Omega(n^{\log_b a + \epsilon})$	$T(n) \in \Theta(f(n))$

4 Definitions

Algorithm Any well-defined computational procedure that takes a set of values as input and produces a set of values as output in a finite number of steps

Correct Algorithm One returns the correct solution for every valid instance of a problem

Loop Invariance Define a key property about the relationship among variables of the algorithms.

- Holds in the *initial* case.

  Initialization The loop invariance must be true prior to the first iteration.

- Is *maintained* each iteration.

Maintenance If the property holds prior to an iteration, it must still hold after the iteration is complete.

- Yields correctness when the loop *terminates*.

  Termination The invariant provides a useful property that helps demonstrate the algorithm is correct.

Heapify Go all the way down to the heap and fix the violations of the max-heap property by sifting-up