Introduction to Algorithms

Note

April 19, 2020

Contents

Part I

Foundations

Chapter 1

The Role of Algorithms in Computing

1. Algorithms: an *algorithm* is any well defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*.

2. Data Structure: a *data structure* is a way to store and organize data in order to facilitate access and modifications.

3. **NP-complete Problems:**

   - Although no efficient algorithm for an NP-complete problem has ever been found, nobody has ever proven that an efficient algorithm for one cannot exist.
   - If an efficient algorithm exists for any one problem, then efficient algorithms exist for all of them.
   - Several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms. Hence a small change to the problem statement can cause a big change to the efficiency of the best known algorithm.

Chapter 2

Getting Started

2.1 Insertion sort

1. **Insertion Sort:**

   - **Pseudocode:**

     INSERTION-SORT(A)
     1 **for** $j = 2$ **to** $A.length$
     2 $key = A[j]$
     3 // Insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$.
     4 $i = j - 1$
     5 **while** $i > 0$ and $A[i] > key$
     6 $A[i+1] = A[i]$
     7 $i = i - 1$
     8 $A[i+1] = key$

   - IN PLACE: it rearranges the nunbers within the array A, with at most a constant nunber of them stored outside the array at any time.
   - **Loop invariants:**
     - Three things about a loop invariant:
       (a) **Initialization:** It is true prior to the first iteration of the loop.
       (b) **Maintenance:** If is true before an iteration of the loop, it remains true before the next iteration.
       (c) **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithms is correct.
     - When the loop is a **for** loop, the moment at which we check the loop invariant just prior to the first iteration is immediately after the initial assignment to the loop-counter variable and just before the first test in the loop header.

2.2 Analyzing algorithms

1. **Random-access machine (RAM) model:**
   - In the *RAM* model, instructions are executed one after another, with no concurrent operations.
   - The *RAM* model constains instructions commonly found in real computers: arithmetic, data movement, and control.
   - Such computers can compute 2^k in one constant-time instruction by shifting the integer 1 by k positions to the left, as long as k is no more than the number of bits in a computer word.
2. **Analysis of insertion sort:** INSERTION-SORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are. To analyze it, we need to define the terms "running time" and "size of input" more carefully:
   - The best notion for *input size* depends on the problem being studies.
   - The *running time* of an algorithm on a particular input is the number of primitive operations or "steps" executed.
3. **Worst-case, average-case, and best-case analysis**
   - **Order of growth:** Consider only the leading term of a formula and at the same time ignore the leading term's constant coefficient.
   - and more ...

2.3 Designing algorithms

2.3.1 The divide-and-conquer approach

1. **Recursive:** Many useful algorithms are *recursive* in structure: to solve a given problem, they call *themselves* recursively one or more times to deal with closely related subproblems.
2. **Divide-and-conquer approach:**
   (a) **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
   (b) **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
   (c) **Combine** the solutions to the subproblems into the solution for the original problem.
3. MERGE SORT:

- **Pseudocode:**

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

- We place at the end of each array a *sentinel* element, which contains a special value ∞ that we use to simplify our code.

2.3.2 Analyzing divide-and-conquer algorithms

1. **Recurrence:** When an algorithm contains a recursive call to itself, we can often describe its running time by a *recurrence*, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs.

2. **Analyzing the total cost represented by a recurrence:**

   (a) Construct a *recursion tree*.

   (b) Compute the total cost of each level of the tree.

   (c) Sum every level's total cost to get the overall total cost.

3. **Recurrence of Merge Sort:**

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1. \end{cases}$$

4. **Time complexity:** $T(n) = \Theta(n \lg n)$

2.4 Algorithms

1. Insertion-Sort
2. Selection-Sort

```
SELECTION-SORT(A)
1  n = A.length
2  for j = 1 to n − 1
3      smallest = j
4      for i = j + 1 to n
5          if A[i] < A[smallest]
6              smallest = i
7      exchange A[j] with A[smallest]
```

3. Merge-Sort
4. Bubble-Sort

```
BUBBLE-SORT(A)
1  n = A.length
2  for i = 1 to n − 1
3      for j = n downto i + 1
4          if A[j] < A[j − 1]
5              exchange A[j] with A[j − 1]
```

5. Linear-Search

```
LINEAR-SEARCH(A, ν)
1  n = A.length
2  for i = 1 to n
3      if ν == A[i]
4          return i
5  return NIL
```

6. Binary-Search

```
BINARY-SEARCH(A, ν, low, high)
1  if low > high
2      return NIL
3  mid = ⌊(low + high)/2⌋
4  if ν == A[mid]
5      return mid
6  elseif ν > A[mid]
7      return BINARY-SEARCH(A, ν, mid + 1, high)
8  else return BINARY-SEARCH(A, ν, low, mid − 1)
```

7. INVERSIONS

```
COUNT-INVERSIONS(A, p, r)
1   inversions = 0
2   if p < r
3       q = ⌊(p + r)/2⌋
4       inversions = inverions + COUNT-INVERSIONS(A, p, q)
5       inversions = inverions + COUNT-INVERSIONS(A, q + 1, r)
6       inversions = inverions + MERGE-INVERSIONS(A, p, q, r)
7   return inversions
```

```
MERGE-INVERSIONS(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  inversions = 0
13  counted = FALSE
14  for k = p to r
15      if counted == FALSE and R[j] < L[i]
16          inversions = inverions + n1 − i + 1
17          counted = TRUE
18      if L[i] ≤ R[j]
19          A[k] = L[i]
20          i = i + 1
21      else A[k] = R[j]
22          j = j + 1
23          counted = FALSE
24  return inversions
```

2.4.1 Summary of sorting algorithms

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

Figure 2.1: Summary of sorting algorithms

Chapter 3

Growth of Functions