Chapter 1: Introduction to Algorithms

Part III

Contents

Proof of correctness of an algorithm

Motivation

Proof of algorithm correctness

1. Gives us more confidence in the correctness of our algorithms
2. Helps us to find subtle errors

Proving algorithm correctness

Basic techniques:

1. Handling **iteration**: using **loop invariants**
2. Handling **recursion**: using **proof by induction**

Proof of correctness using loop invariants

Loop invariant is a condition that is true immediately before and after the loop

To say an algorithm is correct, we must show 3 things about a loop invariant:

1. Initialization: It is true prior to the first iteration of the loop
2. Maintenance: Each iteration maintains the loop invariant
3. Termination: The loop terminates. When it does, the invariant gives us a useful property that helps show that the algorithm is correct.

Proof of correctness using loop invariants

Example:

```
r =  0, i = 0;
while ( i < m ) {
    r = r + a;
    i = i + 1;
}
```

Proof of correctness using loop invariants

Example:

```
r =  0, i = 0;
while ( i < m ) {
    r = r + a;
    i = i + 1;
}
```

Claim: This code computes m.a

Loop invariant: r = i.a

Proof of correctness using loop invariants

Example:

```
r =  0, i = 0;
while ( i < m ) {
    r = r + a;
    i = i + 1;
}
```

Claim: This code computes m.a

Loop invariant: r = i.a

Initialization: Before the loop: i = 0, so r = 0.a ⇒ r = 0, which is true

Maintenance:
Assume r = i.a before the loop
r_{new} becomes r_{old} + a = i.a + a = (i+1).a = i_{new}.a
So, the loop maintains the loop invariant

Termination: Since i grows every iteration, it will be at some point be equal to m. This terminates the loop. So, when the loop is done, i = m. From the invariant, we derive that r = m.a
Q.E.D.

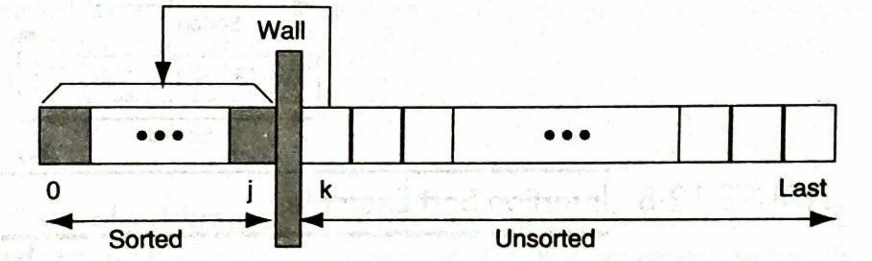
Insertion Sort

One of the most common sorting techniques used by card players. As they pick up each card, they insert it into the proper sequence in their hand.

Main idea:

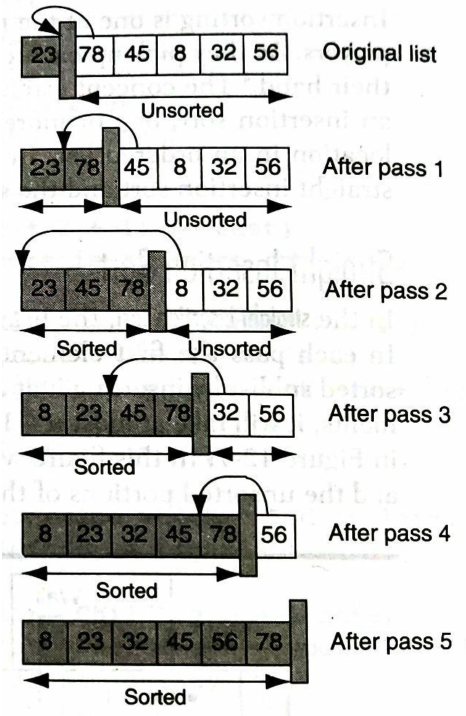
The list is divided into two parts: sorted and unsorted

In each pass of an insertion sort, the first element of the unsorted sublist is inserted into its correct location in the sorted sublist



Insertion sort

Example



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[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Assignment

1. Analyse best case and worst case time complexities of insertion sort.

Proof of correctness of insertion sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. 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
```

Claim: Insertion sort works correctly on array of length n.

Loop invariant:

At the start of each iteration of the for loop of lines 1 - 8, the subarray A[1..j-1] consists of the elements originally in A[1..j-1], but in sorted order

Proof of correctness of insertion sort

Initialization: Before the iteration, j = 2
The subarray A[1..j-1] consists of only A[1], which is the original element in A[1]. This subarray is already sorted.

Maintenance:
The body of the for loop works by moving A[j-1], A[j-2], A[j-3] and so on by one position to the right until it finds the proper position for A[j] (lines 4 - 7), at which point it inserts the value of A[j] (line 8).
The subarray A[1..j] then consists of the elements originally in A[1..j], but in sorted order.
Incrementing j for the next iteration then preserves the loop invariant

Proof of correctness of insertion sort

Termination:
j grows by 1 in each iteration. The loop terminates when j = n+1

From the loop invariant, we can derive that (by substituting n+1 for j)
the subarray A[1..n] consists of the elements originally in A[1..n], but in sorted order.

Since the subarray A[1..n] is the entire array, we conclude that the entire array is sorted.

QED

Proof of correctness using proof by induction

Proof by induction

Suppose that P(n) is a predicate defined on $n \in \{1, 2, \ldots\}$. If we can show that

1. P(1) is true, and
2. $(\ \forall k < n\ \ [P(k)]\ ) \implies P(n)$

Then P(n) is true for all integers $n \geq 1$.

Proof by induction

In other words, to prove some assertion P(n) for all positive numbers $n \geq 1$, we need to prove the induction property (in 2 parts)

1. Base case / a trivial case: P(1)
2. Inductive step:
   Show that if the assertion holds for all smaller values, then it also holds for n, i.e. we assume that for every positive integer $n \geq 2$, P(k) holds for all $k < n$ (this is called **inductive hypothesis**), and then establish that P(n) holds as well.

Quick Sort

Like merge sort, quick sort also uses divide-and-conquer paradigm

Steps:

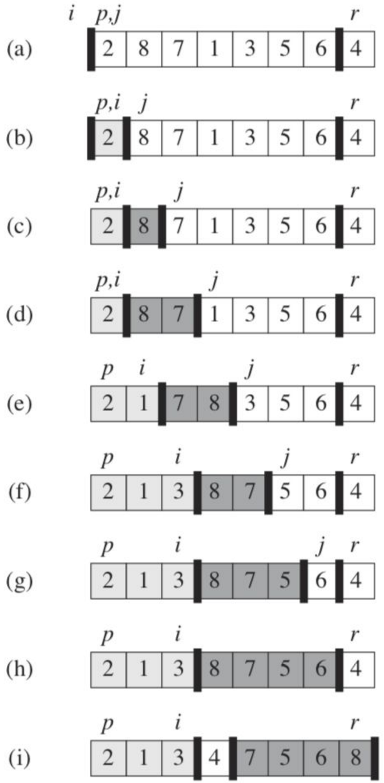
1. **Divide**: Select any element from the list. Call it the **pivot**. Then partition the list into two sublists such that all the elements in the left sublist are less than or equal to the pivot and those of the right sublist are greater than or equal to the pivot
2. **Conquer**: Recursively sort the two sublists
3. **Combine**: Since the subarrays are sorted in place, no work is needed to combine them: the entire list is now sorted.

Quick sort

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

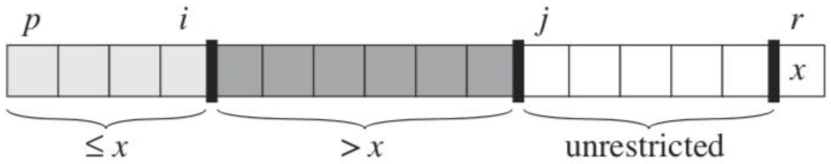
Quick sort

```
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
```



Quick sort

```
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
```



Proof of correctness of Partition algorithm

```
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
```

Loop invariant

For any array index k,

1. If $p \le k \le i$, then $A[k] \le x$,
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$,
3. If $k = r$, then $A[k] = x$

Proof of correctness of Partition algorithm

Initialization:

Prior to the first iteration of the loop, i = p - 1, j = p. Conditions #1 and #2 are trivially satisfied because no values lie between p and i and no values lie between i+1 and j-1.

Condition #3 is satisfied because x is assigned A[r] in line 1.

Loop invariant

For any array index k,

1. If $p \le k \le i$, then $A[k] \le x$,
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$,
3. If $k = r$, then $A[k] = x$

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          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Proof of correctness of Partition algorithm

Maintenance:

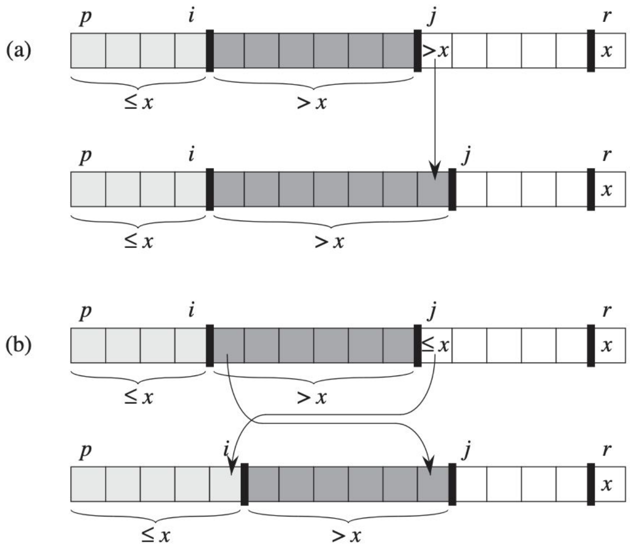


Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \leq x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Loop invariant

For any array 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

```
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
```

Proof of correctness of Partition algorithm

Termination:

The loop terminates when $j = r$.

every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets:

- those less than or equal to x,
- those greater than x, and
- a singleton set containing x.

Loop invariant

For any array index k,

1. If $p \le k \le i$, then $A[k] \le x$,
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$,
3. If $k = r$, then $A[k] = x$

```
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
```

Proof of correctness of Partition algorithm

Line 7 swaps the pivot element, A[r] with the leftmost element greater than x, A[i+1]. If q = i+1, then A[q] is strictly less than every element of A[q+1..r]. Thus the pivot is in its correct place after this line.

Line 8 returns the pivot's new index.

Loop invariant

For any array index k,

1. If $p \le k \le i$, then $A[k] \le x$,
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$,
3. If $k = r$, then $A[k] = x$

```
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          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Assignment

2. Write the recurrence equations for best case and worst case of quick sort and solve them.
   Hint: In the best case, the partition() always picks the middle element as pivot, resulting in two subproblems, each of size no more than n/2. In the worst case, the list already is ordered but in reverse order (this produces one subproblem with n - 1 elements and one with 0 element).

Proof by induction

Example:

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

P(n) = Quicksort is always correct on inputs of length n

Base case P(1) holds (input of length 1)

Inductive hypothesis: P(k) holds for k < n

Proof by induction

Example:

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

Recall that quick sort partitions the input array into 3 parts:

1. The first subarray of size k,
2. The pivot (which will be in its correct place),
3. The remaining subarray of size n−k−1

Proof by induction

Example:

```
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)
```

Recall that quick sort partitions the input array into 3 parts:

1. The first subarray of size k,
2. The pivot (which will be in its correct place),
3. The remaining subarray of size n−k−1

Since k < n, and n-k-1 < n, we can apply the inductive hypothesis to imply that the two recursive calls (line 3-4) will correctly sort these two subarrays.

Thus, putting the first subarray, the pivot and the second subarray together will result in the sorted array. Thus, the quicksort algorithm is correct.

Assignment

1. Analyse best case and worst case time complexities of insertion sort.
2. Write the recurrence equations for best case and worst case of quick sort and solve them.
   Hint: In the best case, the partition() always picks the middle element as pivot, resulting in two subproblems, each of size no more than n/2. In the worst case, the list already is ordered but in reverse order (this produces one subproblem with n - 1 elements and one with 0 element).
3. Prove the correctness of merge sort algorithm. You will have to prove the correctness of **merge** algorithm using loop invariants, and that of recursive **merge sort** algorithm using proof by induction.