Fall 2019

Algorithms

#1

Divide & Conquer

)ffice : 47 Ahmed Kamha St. Camp Caesar Alexandria Egypt
+2035933145 Fax:+2035912212 Mob :+201000171795

Insertion Sort

Proof of Correctness

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
```

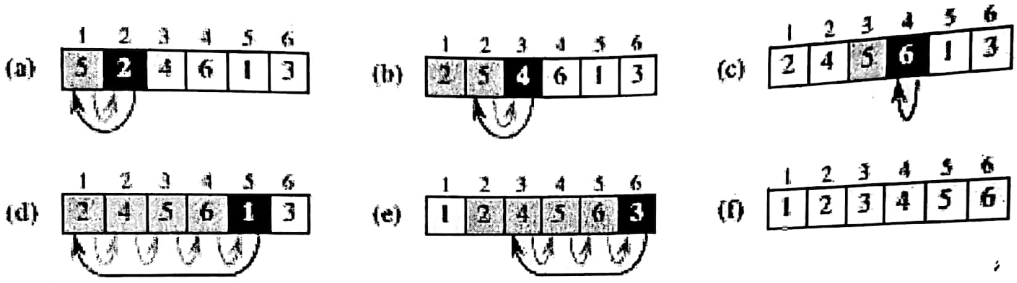


Figure 2.2 The operation of INSERTION-SORT on the array $A = (5, 2, 4, 6, 1, 3)$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. (a)–(e) The iterations of the for loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. (f) The final sorted array.

Loop invariants and the correctness of insertion sort

In order to *prove* the correctness of an algorithm, i.e. that it will work for *any* input set, we construct a *loop invariant* which is a condition that holds for each iteration of the loop (thus the name invariant). Then the proof consists of verifying that the algorithm maintains the loop invariant for three conditions:

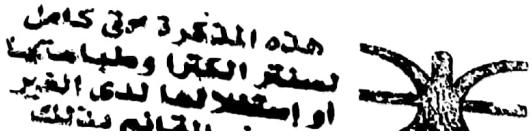
- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

The loop invariant for insertion sort can be stated as follows:

→ At each step, A [1...j-1] contains the first j-1 elements in SORTED order.

Let us see how these properties hold for insertion sort.

Initialization: Prior to the loop $j = 2 \Rightarrow A[1..j\text{-}1] = A[1]$ which contains only the $A[1..j\text{-}1]$ elements (of which there is only one) and since there is only a single element they are trivially sorted.



Maintenance: The outer **for** loop selects element A [j] and positions it properly into $A[1..j\text{-}1]$ via the while loop. Since the array A $[1...j\text{-}1]$ began sorted, inserting element A[j] into the proper place produces $A[1..j]$ in sorted order (and contains the first j elements).

Termination: The loop terminates when $j = n+1$
$\Rightarrow A[1..j\text{-}1] = A[1..(n+1)\text{-}1] = A[1..n]$ which since the array remains sorted after each iteration gives $A[1..n]$ is sorted when the loop terminates (and contains *all* the original elements) $\Rightarrow$ the entire *original* array is sorted. Hence the algorithm is correct

Divide and Conquer Algorithms

In divide-and-conquer, we solve a problem recursively, applying three steps at each level of the recursion:

Divide the problem into a number of sub problems that are smaller instances of the same problem.

Conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner.

Combine the solutions to the sub problems into the solution for the original problem.

1. Merge Sort

The **merge sort** algorithm closely follows the divide-and-conquer paradigm. Intuitively,
it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to product the sorted answer.

Merge Sort

A sorting algorithm based on **divide** and **conquer**.
Because we are dealing with sub problems we state the sub problem as sorting a subarray A[p…r]. Initily p=1 and r=n but this values change as we recurse through sub problems to sort A[p…r]

Divide by splitting into two subarrays $A[p..q]$ and $A[q+1..r]$, where q is tl halfway point of $A[p..r]$.

Conquer by recursively sorting the two subarrays $A[p..q]$ and $A[q+1..r]$.

Combine by merging the two sorted subarrays $A[p..q]$ and $A[q+1..r]$ to pr duce a single sorted subarray $A[p..r]$. To accomplish this step, we'll define procedure MERGE(A, p, q, r).

The recursion bottoms out when the subarray has just 1 element, so that it's trivial sorted.

```
MERGE-SORT(A, p, r)
if p < r                          ▷ Check for base case
  then q ← ⌊(p + r)/2⌋            ▷ Divide
       MERGE-SORT(A, p, q)        ▷ Conquer
       MERGE-SORT(A, q + 1, r)    ▷ Conquer
       MERGE(A, p, q, r)          ▷ Combine
```

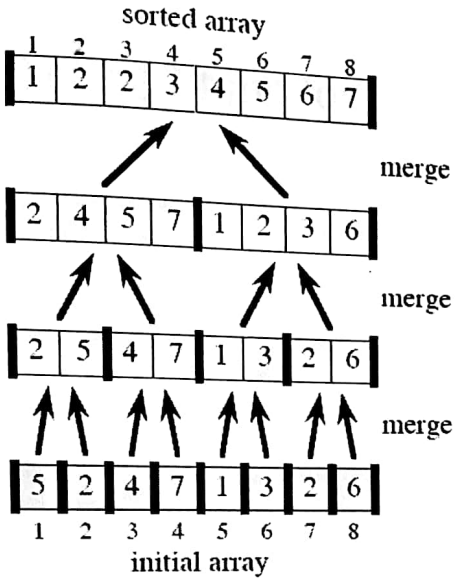
Initial call: MERGE-SORT$(A, 1, n)$

Merge Algorithm that merges two sorted lists into one sorted List.

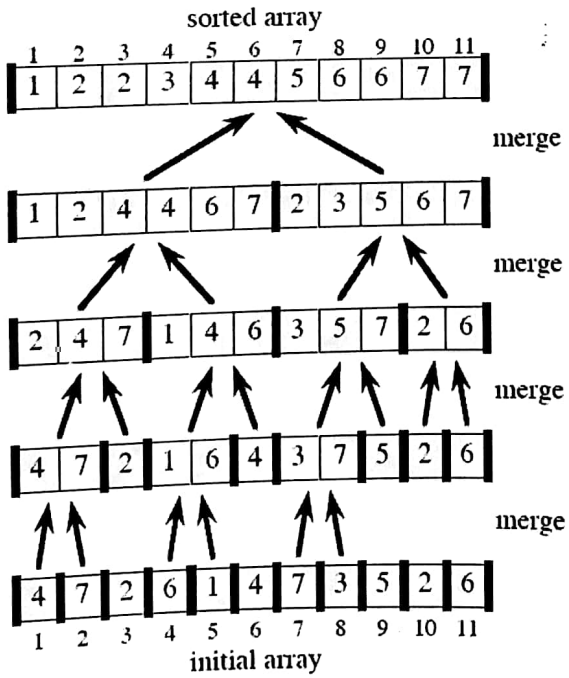
```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
    do L[i] ← A[p + i − 1]
for j ← 1 to n2
    do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
          then A[k] ← L[i]
               i ← i + 1
          else A[k] ← R[j]
               j ← j + 1
```

Example for merge sort

Example: Bottom-up view for $n = 8$: *[Heavy lines demarcate subarrays used in subproblems.]*



Bottom-up view for $n = 11$:



Analyzing merge sort running time

For simplicity assume n is power of 2 →the base case occurs when n=1

When n≥2 time for merge sort steps

Divide: just compute q as the average of p and r → O(1)

Conquer: Recursively solve 2 sub problems each of size n/2

Combine: Merge n element sub array takes O(n)

So we write the merge sort recurrence as following:

$T(n) = 2T(n/2) + n$

$T(1) = 1$

By Solving this recurrence so the merge sort algorithm is **O (n log n)**

The maximum-subarray problem

You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.

For example,
if the given array is {-2, -5, **6, -2, -3, 1, 5**, -6},
Then the maximum subarray sum is 7 (6,-2,-3,1,5).

Solution

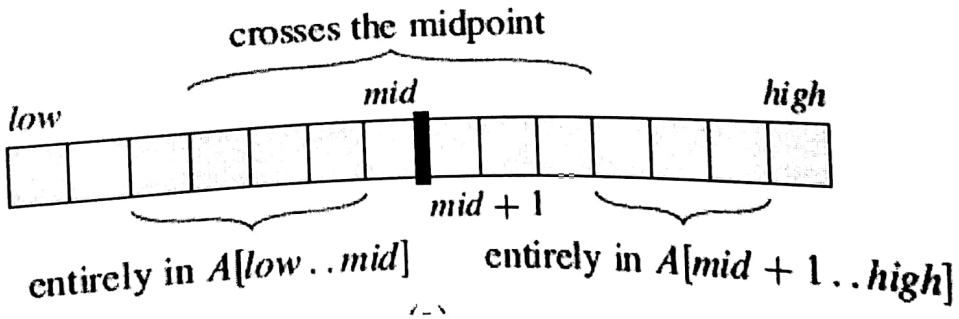
We can easily devise a brute-force solution to this problem:
The solution is to run two loops. The outer loop picks the beginning element, the inner loop finds the maximum possible sum with first element picked by outer loop and compares this maximum with the overall maximum. Finally return the overall maximum. The time complexity of this solution is **O (n^2).**

Using **Divide and Conquer** approach, we can find the maximum subarray sum in O (nLogn) time. Following is the Divide and Conquer algorithm.

1) Divide the given array in two halves

2) Return the maximum of following three

- Maximum subarray sum in left half (Make a recursive call)
- Maximum subarray sum in right half (Make a recursive call)
- Maximum subarray sum such that the subarray crosses the midpoint



We can easily find a maximum subarray crossing the midpoint in time linear in the size of the subarray **A**[***low*...*high***]. This problem is *not* a smaller instance of our original problem, because it has the added restriction that the subarray it chooses must cross the midpoint

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

The above procedure works as follows.

Find a maximum subarray of the left half, **A[low…mid]**. Since this subarray must contain A[mid], the **for** loop of starts the index i at mid and works down to low, so that every subarray it considers is of the form **A[i…. mid]**

initialize the variables left-sum, which holds the greatest sum found so far and sum is holding the sum of the entries in **A[i ….mid**] Whenever we find, a subarray **A[i…mid]** with a sum of values greater than left-sum, we update left-sum to this subarray's sum and we update the variable max-left to record this index

Then the algorithm will work analogously for the right half, **A[mid+1…high]**.
Here, the **for** loop starts the index j at mid+1 and works up to high, so that every subarray it considers is of the form **A[mid +1…. j]**.

Finally returns the indices *max-left* and *max-right* that demarcate a maximum subarray crossing the midpoint, along with the sum *left-sum* + *right-sum* of the values in the subarray A[*max-left …..max-right*].

If the subarray A[*low…..high]* contains n entries (so that n= *high*+ *low*- 1),

We claim that the call FIND-MAX-CROSSING-SUBARRAY takes **O(n)** time.

With a linear-time **FIND-MAX-CROSSING-SUBARRAY** procedure in hand, we can write pseudocode for a divide-and-conquer algorithm to solve the maximum subarray problem:

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2       return (low, high, A[low])          // base case: only one elemer
 3  else mid = ⌊(low + high)/2⌋
 4       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8            return (left-low, left-high, left-sum)
 9       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10            return (right-low, right-high, right-sum)
11       else return (cross-low, cross-high, cross-sum)
```

Analyzing the divide-and-conquer algorithm

we can write the recurrence as following:

$T(n) = 2T(n/2) + n$

$T(1) = 1$

By Solving this recurrence so the merge sort algorithm is **O (n log n)**

Matrix Multiplication

Given two square matrices A and B of size n x n each, find their multiplication matrix.

Normal Solution

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

Time Complexity of above method is $O(n^3)$.

A simple Divide and Conquer Solution

Following is simple Divide and Conquer method to multiply two square matrices.

1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.

2) Calculate following values recursively.

ae + bg

af + bh

ce + dg

cf + dh.

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}$$

A, B and C are square metrices of size N x N

a, b, c and d are submatrices of A, of size N/2 x N/2

e, f, g and h are submatrices of B, of size N/2 x N/2

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(n^2)$ time. So the time complexity can be written as

$$T(N) = 8T(N/2) + O(N^2)$$

From Master's Theorem, time complexity of above method is $O(N^3)$

Thus, this simple divide-and-conquer approach is no faster than the straightforward approach

Strassen's Method

The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

$$p1 = a(f - h) \qquad p2 = (a + b)h$$
$$p3 = (c + d)e \qquad p4 = d(g - e)$$
$$p5 = (a + d)(e + h) \qquad p6 = (b - d)(g + h)$$
$$p7 = (a - c)(e + f)$$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$$T(N) = 7T(N/2) + O(N^2)$$

From Master's Theorem, time complexity of above method is $O(N^{Log7})$ which is approximately $O(N^{2.8074})$

Multiplying two large numbers

Let's say we need to multiply X*Y

We will divide x into two halves and Y as well

$X = X_L * 2^{n/2} + X_R$

[X_L and X_R contain leftmost and rightmost n/2 bits of X]

$Y = Y_L * 2^{n/2} + Y_R$

[Y_L and Y_R contain leftmost and rightmost n/2 bits of Y]

For Example Let Say X is 11010011

So X =1101*2^4+0011

The product XY can be written as following.

$$XY = (X_L * 2^{n/2} + X_R)(Y_L * 2^{n/2} + Y_R)$$

$$= 2^n X_L Y_L + 2^{n/2}(X_L Y_R + X_R Y_L) + X_R Y_R$$

If we take a look at the above formula, there are four multiplications of size n/2, so we basically divided the problem of size n into four sub-problems of size n/2. But that doesn't help because solution of recurrence

$T(n) = 4T(n/2) + O(n)$ is $O(n^2)$

The tricky part of this algorithm is to change the middle two terms to some other form so that only one extra multiplication would be sufficient. The following is tricky expression for middle two terms.

$X_L\ Yr + XrY_L = (X_L + Xr)(\ Y_L + Yr) - X_L Y_L - XrYr$

So the final value of XY becomes

$XY = 2n\ X_L\ Y_L + 2n/2 * [(X_L + X_R)(\ Y_L + Yr) - X_L\ Y_L - X_R\ Y_R] + X_R\ Y_R$

With above trick, the recurrence becomes

$T(n) = 3T(n/2) + O(n)$ and solution of this recurrence is $O(n^{1.59})$.