Analysis and Design of Algorithms
Divide and Conquer

Angel Napa

April 2024

1 Maximum Subarray Problem

- Input: Array $A[1..n]$ of integers (not necessarily positive).
- Output: Indices i, j such that the sum of the elements in $A[i..j]$ is maximum possible.

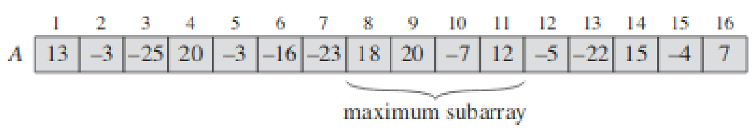

Figure 1: Taken from the book Cormen, Introduction to Algorithms

Brute force solution: Try all possible subsequences. As there are $\binom{n}{2}$ possibilities, the algorithm is $\Theta(n^2)$.

Can we do better?

Note that, given an array $A[low..high]$, a maximum subarray has three possibilities. It is

- Entirely in $A[low..mid]$
- Entirely in $A[mid + 1..high]$
- With a part in $A[low..mid]$ and the other part in $A[mid + 1..high]$

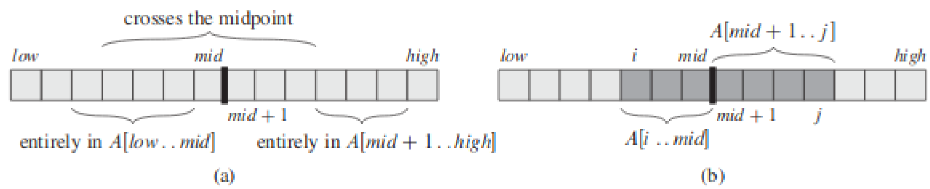

Figure 2: Taken from the book Cormen, Introduction to Algorithms

This observation allows us to design a divide and conquer algorithm for the problem.

- **Divide** We split A into two subarrays of size $(high - low + 1)/2$
- **Conquer**: In each subarray, we find the corresponding maximum subarray.
- **Combine** We find which of the two subarrays from the recursive calls has the maximum sum.

  Then we need to compare this sum with a third candidate, which is the maximum subarray with a part in $A[low..mid]$ and the other part in $A[mid+1..high]$

Next, we will see how to find this mentioned candidate in the "Combine" operation.

Input: An array A of integers, and three indices $low \leq mid < high$.

Output: Three integers $i, j, \sum_{k=i}^{j} A[k]$ such that $\sum_{k=i}^{j} A[k]$ is maximum for all i, j satisfying $i \leq mid < j$

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

Figure 3: Taken from the book Cormen, Introduction to Algorithms

Execution time:

$T(n) = c_1 + (low - mid + 1)c_2 + (high - mid)c_3 = c_1 + c_4(low - mid + 1) = c_1 + 4n = \Theta(n)$

Now we will analyze the main algorithm.

Input: An array of integers $A[low..high]$.

Output: Three integers $i, j, \sum_{k=i}^{j} A[k]$ such that $\sum_{k=i}^{j} A[k]$ is maximum for all i, j

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
1   if high == low
2       return (low, high, A[low])          // base case: only one element
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)
```

Figure 4: Taken from the book Cormen, Introduction to Algorithms

Execution time of FIND-MAXIMUM-SUBARRAY. When $n = 1$, lines 1 and 2 are executed: time $c_1 + c_2$. When $n > 1$, we have $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_3 + c_7 + c_8 + c_9 + c_{10} + c_{11} + k_1 n$

$$T(n) = \begin{cases} c & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + k_1 n + k_2 & \text{otherwise} \end{cases}$$

So $T(n) = \Theta(n \log n)$. (To see this, apply the master theorem, since when n is a power of 2, we have $T(n) = 2T(n/2) + kn$), we have $a = b = 2, k = 1$, second case of the master theorem)

2 Multiplication of Natural Numbers

- Input: Two natural numbers a and b with n digits each.
- Output: The product $a \cdot b$

The usual algorithm:

9999	A
7777	B
69993	C
69993	D
69993	E
69993	F
77762223	G

In the previous figure, $A \cdot B = C + D + E + F = G$. The execution time is proportional to $4 + 4 + 4 + 4 = 16 = 4^2$.

This is a simple algorithm that can be described as follows:

Require: Two integers represented by $a[1..n], b[1..n]$
Ensure: The product $a \cdot b$

BASIC-MULTIPLICATION(a, b, n)	*cost*	*times*
total = 0	c_1	1
for $j = 1$ **to** n	c_2	$n + 1$
sum = 0	c_3	n
for $i = 1$ **to** n	c_4	$(n + 1) \cdot n$
sum = sum $\cdot$ 10 + $b[j] \cdot a[i]$	c_5	$n \cdot n$
total = total $\cdot$10 + sum	c_6	n
return total	c_7	1

It's clear that the execution time is $\Theta(n^2)$. Now we'll see how to improve this algorithm.

2.1 A Divide and Conquer Algorithm

Note that, given two natural numbers a and b with n digits, we can express them as

$$a = a_1 \cdot 10^m + a_2,$$

$$b = b_1 \cdot 10^m + b_2.$$

Where $m = \lceil n/2 \rceil$.

For example, $3213209842 = 32132 \cdot 10^5 + 09842$ or $953421412 = 9534 \cdot 10^5 + 21412$.

Therefore

$$\begin{aligned} a \cdot b &= (a_1 \cdot 10^m + a_2) \cdot (b_1 \cdot 10^m + b_2) \\ &= (a_1 b_1) \cdot 10^{2m} + (a_1 b_2 + a_2 b_1) \cdot 10^m + (a_2 b_2) \end{aligned}$$

This way, we can divide our original problem into four subproblems: multiplying a_1 by b_1, multiplying a_1 by b_2, multiplying a_2 by b_1, and multiplying a_2 by b_2.

We obtain the following divide and conquer algorithm:

Require: Two integers a and b with n digits, where n is a power of two, and both a and b do not contain zeros.
Ensure: The product $a \cdot b$

MULTIPLICATION-DC (a, b, n)	*cost*	*times*
if $n = 1$	$\Theta(1)$	1
return $a \cdot b$	$\Theta(n)$	1
$a_1 = \lfloor a/10^{n/2} \rfloor$	$\Theta(n)$	1
$a_2 = a \mod 10^{n/2}$	$\Theta(n)$	1
$b_1 = \lfloor b/10^{n/2} \rfloor$	$\Theta(n)$	1
$b_2 = b \mod 10^{n/2}$	$\Theta(n)$	1
$p =$ MULTIPLICATION-DC$(a_1, b_1, n/2)$	$T(n/2)$	1
$q =$ MULTIPLICATION-DC$(a_1, b_2, n/2)$	$T(n/2)$	1
$r =$ MULTIPLICATION-DC$(a_2, b_1, n/2)$	$T(n/2)$	1
$s =$ MULTIPLICATION-DC$(a_2, b_2, n/2)$	$T(n/2)$	1
return $p \cdot 10^n + (q + r) \cdot 10^{n/2} + s$	$\Theta(n)$	1

Execution time:

$$T(n) = \begin{cases} \Theta(n) & n = 1 \\ 4T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

By the Master Theorem, $\lg 4 / \lg 2 = 2 > 1$. Thus $T(n) = \Theta(n^2)$.

We see that MULTIPLICATION-DC does not improve BASIC-MULTIPLICATION.

2.2 Karatsuba's Algorithm

This algorithm will improve the previous recurrence by making only 3 recursive calls. Therefore, we'll have an execution time of $\Theta(n^{\lg 3/\lg 2}) = \Theta(n^{1.59})$.

The main observation is as follows.

Given two natural numbers a and b with n digits, we express them as in the previous subsection:

$$a = a_1 \cdot 10^m + a_2,$$

$$b = b_1 \cdot 10^m + b_2.$$

Where $m = \lceil n/2 \rceil$.

We have

$$\begin{aligned} a \cdot b &= (a_1 \cdot 10^m + a_2) \cdot (b_1 \cdot 10^m + b_2) \\ &= (a_1 b_1) \cdot 10^{2m} + (a_1 b_2 + a_2 b_1) \cdot 10^m + (a_2 b_2) \\ &= (a_1 b_1) \cdot 10^{2m} + ((a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2) \cdot 10^m + (a_2 b_2) \end{aligned}$$

This way, we only need to calculate three products: $a_1 b_1$, $a_2 b_2$, and $(a_1 + a_2)(b_1 + b_2)$. The product $a_1 b_2 + a_2 b_1$ can be calculated as $(a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2$.

Require: Two integers a and b with n digits, where n is a power of 2 and both a and b do not contain zeros

Ensure: The product $a \cdot b$

KARATSUBA (a, b)	*cost*	*times*
if $n \leq 1$	$\Theta(1)$	1
return $a \cdot b$	$\Theta(n)$	1
$a_1 = \lfloor a/10^{n/2} \rfloor$	$\Theta(n)$	1
$a_2 = a \mod 10^{n/2}$	$\Theta(n)$	1
$b_1 = \lfloor b/10^{n/2} \rfloor$	$\Theta(n)$	1
$b_2 = b \mod 10^{n/2}$	$\Theta(n)$	1
$p = \text{KARATSUBA}(a_1, b_1)$	$T(n/2)$	1
$q = \text{KARATSUBA}(a_1 + a_2, b_1 + b_2)$	$T(n/2)$	1
$s = \text{KARATSUBA}(a_2, b_2)$	$T(n/2)$	1
return $p \cdot 10^n + (q - p - s) \cdot 10^n + s$	$\Theta(n)$	1

Execution time:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 3T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

By the Master Theorem, $\lg 3/\lg 2 > 1$. Thus $T(n) = \Theta(n^{\lg 3/\lg 2}) = \Theta(n^{1.59\cdots})$.

Observation: In the above pseudocode, we have assumed for simplicity that both a and b have n digits and that n is a power of 2. Otherwise, a sufficient trick for a good implementation is to pad with leading zeros if necessary.

3 Matrix Multiplication

Problem: Given two matrices $A = (a_{ij})$ and $B = (b_{ij})$ of dimensions $n \times n$, calculate $C = A \cdot B$. Recall that $C = (c_{ij})$ is defined as

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

We have the following algorithm to compute C.

Input: Two matrices A and B of dimensions $n \times n$ Output: $A \cdot B$

MULTIPLY (A, B)

```
for i = 1 to n
  for j = 1 to n
    c_ij = 0
    for k = 1 to n
      c_ij = c_ij + a_ik b_kj
return C
```

A simple analysis of the algorithm tells us that its execution time is $\Theta(n^3)$.

3.1 Divide and Conquer Algorithm

For simplicity, let's assume that n is a power of 2. We can partition each of the matrices into four parts. That is,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Therefore,

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \tag{1}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \tag{2}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \tag{3}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \tag{4}$$

From these equations, we can formulate the following divide and conquer algorithm.

Input: Two matrices A and B of dimensions $n \times n$ Output: $A \cdot B$

Multiply-DC (A, B)

if $n = 1$
    $c_{11} = a_{11} \cdot b_{11}$
else
    Create auxiliary matrices
    C_{11}=Multiply-DC(A_{11}, B_{11})+ Multiply-DC(A_{12}, B_{21})
    C_{12}=Multiply-DC(A_{11}, B_{12})+ Multiply-DC(A_{12}, B_{22})
    C_{21}=Multiply-DC(A_{21}, B_{11})+ Multiply-DC(A_{22}, B_{21})
    C_{22}=Multiply-DC(A_{21}, B_{12})+ Multiply-DC(A_{22}, B_{22})
    Fill C from its parts
return C

We have the following recurrence.

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{otherwise} \end{cases}$$

The Master Theorem tells us that $T(n) = \Theta(n^3)$. It has not improved the basic algorithm.

3.2 Strassen's Algorithm

Analogously to the Karatsuba algorithm, Strassen's algorithm attempts to perform only 7 recursive operations. Before the calls, we will create the following matrices:

$$S_1 = B_{12} - B_{22}$$

$$S_2 = A_{11} + A_{12}$$
$$S_3 = A_{21} + A_{22}$$
$$S_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{22}$$
$$S_6 = B_{11} + B_{22}$$
$$S_7 = A_{12} - A_{22}$$
$$S_8 = B_{21} + B_{22}$$
$$S_9 = A_{11} - A_{21}$$
$$S_{10} = B_{11} + B_{12}$$

Input: Two matrices A and B of dimensions $n \times n$ Output: $A \cdot B$

STRASSEN (A, B)

if $n = 1$
    $c_{11} = a_{11} \cdot b_{11}$
else
    Create auxiliary matrices
    P_1=MULTIPLY-DC(A_{11}, S_1)
    P_2=MULTIPLY-DC(S_2, B_{22})
    P_3=MULTIPLY-DC(S_3, B_{11})
    P_4=MULTIPLY-DC(A_{22}, S_4)
    P_5=MULTIPLY-DC(S_5, S_6)
    P_6=MULTIPLY-DC(S_7, S_8)
    P_7=MULTIPLY-DC(S_9, S_{10})
    C_{11}=$P_5 + P_4 - P_2 + P_6$
    C_{12}=$P_1 + P_2$
    C_{21}=$P_3 + P_4$
    C_{22}=$P_5 + P_1 - P_3 - P_7$
    Fill C from its parts
return C

We have the following recurrence

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{otherwise} \end{cases}$$

The Master Theorem tells us that $T(n) = \Theta(n^{\lg 7})$.

4 Counting Inversions

- Input: Array $A[1..n]$ of distinct integers
- Output: Number of inversions. Where an *inversion* is a pair (i, j) such that $i < j$ and $A[i] > A[j]$

For example, let $A = [2, 4, 1, 3, 5]$. The inversions are $(1, 3), (2, 3), (2, 4)$

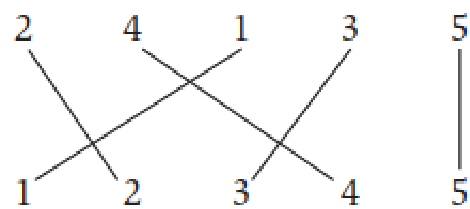

Figure 5: Taken from the book Kleinberg-Tardos, Algorithm Design

Naive Algorithm: Evaluate all possible ordered pairs and check if they are inversions.

Input: An array of distinct integers $A[1..n]$

Output: The number of inversions in A.

NAIVE-INVERSIONS(A, n)	*cost*	*times*
1: total = 0	c_1	1
2: **for** $i = 1$ to $n - 1$	c_2	n
3: **for** $j = i + 1$ to n	c_3	$\sum_{i=1}^{n-1} n - i + 1$
4: **if** $A[i] > A[j]$	c_4	$\sum_{i=1}^{n-1} n - i$
5: total = total + 1	c_5	$\sum_{i=1}^{n-1} t_i$
6: **return** total	c_6	1

Where t_i is the number of times we execute line 5 during the i-th iteration. Clearly we have $0 \leq t_i \leq n - i$.
This algorithm, both in the worst and best case, consumes $\Theta(n^2)$ time. (An example of the worst case occurs if the array is sorted in decreasing order). Next, we will see a divide and conquer algorithm for this problem.

4.1 Divide and Conquer

We observe that, given the array $A[1..n]$, an inversion (i, j) can be in exactly one of these possibilities:

- $i, j \in \{1, \ldots, \lfloor n/2 \rfloor\}$
- $i, j \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$
- $i \in \{1, \ldots, \lfloor n/2 \rfloor\}$, $j \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$

That gives us the idea of a divide and conquer algorithm.

- **Divide** We divide A into two subarrays of sizes $\lfloor n/2 \rfloor, \lceil n/2 \rceil$
- **Conquer**: In each subarray, we find the number of inversions
- **Combine** We add this number with the number of inversions that have one element before or equal to $\lfloor n/2 \rfloor$ and the other after $\lfloor n/2 \rfloor$. This would give us the total number of inversions.

So, informally, we see that $T(n) = 2T(n/2) + C(n)$, where $C(n)$ is the time to combine. If we want the recurrence to be $\Theta(n \lg n)$ then $C(n)$ must be $\Theta(n)$.

But if we try to count the (i, j) such that $1 \leq i \leq \lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 1 \leq j \leq n$ with brute force, we can spend time proportional to $n/2 \cdot n/2 = \Theta(n^2)$. We need a more efficient algorithm.

Note that if the subarrays $A[1..\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1 \ldots n]$ were already sorted, then we would only spend time $\Theta(n)$ due to the following observation:

If (i, j) is an inversion with $i \in \{1, \ldots, \lfloor n/2 \rfloor\}$ and $j \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$ then (i', j) is also an inversion for all $i' > i$ (5)

To prove (5), it suffices to observe that $A[i] > A[j]$ because (i, j) is an inversion and that $A[i'] > A[i]$ because A is sorted. The following subroutine takes care of this counting. It reminds us of the Merge function of Mergesort. Then, the following subroutine would do what is asked.

Input: An array of distinct integers $A[1..n]$ and three indices p, q, r such that $A[p..q]$ and $A[q+1..r]$ are sorted.

Output: The number of inversions (i, j) in A such that $i \in \{p, \ldots, q\}$, $j \in \{q+1, \ldots, r\}$. Also, it sorts the array $A[p..r]$.

```
Central-Inversions(A, p, q, r)
n1 = q - p + 1
n2 = r - q
Let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
L[n2 + 1] = ∞
i = 1
j = 1
total = 0
for k = p to r
    if L[i] > R[j]
        A[k] = R[j]
        total = total + (n1 + 1 - i)
        j = j + 1
    else
        A[k] = L[i]
        i = i + 1
return total
```

Note that Central-Inversions is essentially the Merge subroutine of Mergesort, which consumes time $\Theta(n)$.

Finally, with the help of this subroutine, we design our main algorithm.

Input: An array of distinct integers $A[p..r]$

Output: The number of inversions in A.

INVERSIONS-DC(A, p, r)	*cost*	*times*
if $(p == r)$	c_1	*1*
return 0	c_2	*0*
$q = \left\lfloor \frac{r-p+1}{2} \right\rfloor$	c_3	1
$total_1 =$ INVERSIONS-DC(A, p, q)	$T(\lfloor n/2 \rfloor)$	1
$total_2 =$ INVERSIONS-DC$(A, q+1, r)$	$T(\lceil n/2 \rceil)$	1
$total_3 =$ CENTRAL-INVERSIONS(A, p, q, r)	kn	1
return $total_1 + total_2 + total_3$	c_5	1

Note that $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + kn$. Then, by the Master Theorem, $T(n) = \Theta(n \lg n)$.