Design and Analysis of Algorithms CS218M

Divide and Conquer Algorithms (2)

Paritosh Pandya

Indian Institute of Technology, Bombay

Autumn, 2022

Maximum Subarray Problem

(Reference CLRS 4.1)

Problem: Find slice of array whose sum of elements is maximum amongst all slices.

E.g. *FIND_MAXIMUM_SUBARRAY*$(A, 1, n)$ gives

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

Maximum Subarray Problem

(Reference CLRS 4.1)

Problem: Find slice of array whose sum of elements is maximum amongst all slices.

E.g. $FIND\_MAXIMUM\_SUBARRAY(A, 1, n)$ gives

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray (positions 8–11)

$i, j, S = FIND\_MAXIMUM\_SUBARRAY(A, low, high)$

Let $sum(A, i, j) \;=\; \Sigma_{k=i}^{k=j} A[k]$

pre $1 \leq low \leq high \leq n$

post $low \leq i \leq j \leq high \wedge S = sum(i, j) \;\; \wedge$
$\forall (k, l) : low \leq k \leq l \leq high. \;\; sum(A, k, l) \leq S$

Naive Solution

Divide and Conquer Algorithm

•

Divide and Conquer Algorithm

- Split array slice (*low*, *high*) in two halves at *mid*

Divide and Conquer Algorithm

- Split array slice (*low*, *high*) in two halves at *mid*
- Recursively, find Maximum slices in left and right halves.

Divide and Conquer Algorithm

- Split array slice (*low*, *high*) in two halves at *mid*
- Recursively, find Maximum slices in left and right halves.
- Choose the maximum of the two.

Divide and Conquer Algorithm

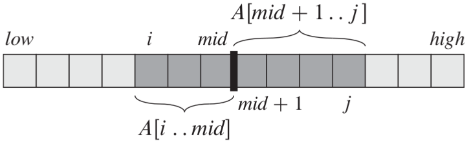
- Split array slice $(low, high)$ in two halves at mid
- Recursively, find Maximum slices in left and right halves.

- Find also the maximum slice (i, j) crossing the mid point, s.t. $low \leq i \leq mid$ and $mid + 1 \leq j \leq high$

- Split array slice (*low*, *high*) in two halves at *mid*
- Recursively, find Maximum slices in left and right halves.

- Find also the maximum slice (i, j) crossing the mid point, s.t. $low \leq i \leq mid$ and $mid + 1 \leq j \leq high$

  $FIND\_MAX\_CROSSING\_SUBARRAY(A, low, mid, high)$ gives



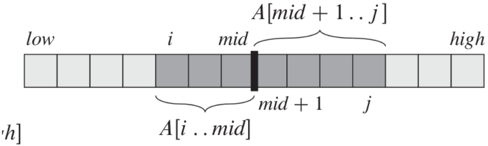
'h]

Divide and Conquer Algorithm

- Split array slice (*low*, *high*) in two halves at *mid*
- Recursively, find Maximum slices in left and right halves.

- Find also the maximum slice (i, j) crossing the mid point, s.t.$low \leq i \leq mid$ and $mid + 1 \leq j \leq high$

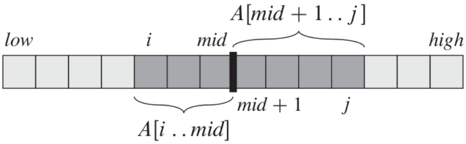
  $FIND\_MAX\_CROSSING\_SUBARRAY(A, low, mid, high)$ gives



- Choose tha maximum of the three slices.
  This gives the maximum subarray within (*low*, *high*). (Why?)

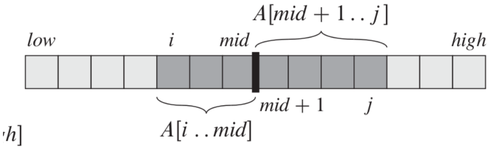
Divide and Conquer Algorithm

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋                                    divide
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)                     conquer
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)                Conquer
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)          Combine
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum      Comb
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```



h]

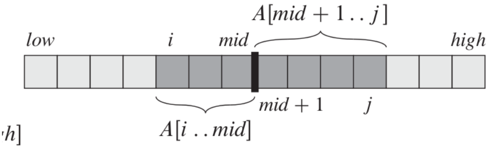
Case 3: Divide and Conqueer



Strategy

We choose *i* independantly so that highest *leftsum* .

Case 3: Divide and Conqueer

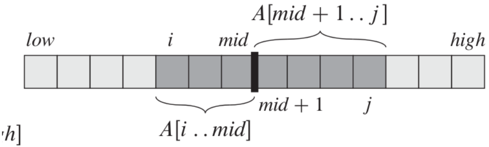


Strategy

We choose *i* independantly so that highest *leftsum* .

```
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
```

Case 3: Divide and Conqueer



Strategy

We choose *i* independantly so that highest *leftsum* .

```
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
```

post: $leftsum = sum(A, i, mid) \;\wedge$

$$\forall k : \; low \leq k \leq mid . sum(A, j, mid) \leq leftsum$$

Case 3: Divide and Conqueer

```
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)
```

Complexity of *FIND_MAX_CROSSING_SUBARRAY*$(A, low, mid, high)$

Let $n = high - low$ and $low \leq mid \leq high$.

```
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)
```

$\Theta(1)$

$\Theta(mid - low)$

$\Theta(1)$

$\Theta(high - m + 1)$

$\Theta(n)$

$\Theta(1)$

Complexity of *FIND_MAXIMUM_SUBARRAY*(*A*, *low*, *high*)

Let $n = high - low$ T(n)

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low                                            } Θ(1)
    return (low, high, A[low])                            }      // base case: only one element
else mid = ⌊(low + high)/2⌋                                 Θ(1)
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)                  T(n/2)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)             T(n/2)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)       Θ(n)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum        }
        return (left-low, left-high, left-sum)              }
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum   } Θ(1)
        return (right-low, right-high, right-sum)           }
    else return (cross-low, cross-high, cross-sum)
```

Recurrence

$$
\begin{aligned}
T(1) &= c_1 \\
T(n) &= 2 * T(n/2) + c_2 * n \qquad \text{for } n > 1
\end{aligned}
$$

Solution

$n^{\log_b a} = n^{\log_2(2)} = n$

$\#leaf = \Theta(f(n))$ case

$n^{\log_b a} * \log_b n = n \times \lg(n)$

Counting Inversions in Array

Inversion is (i, j) s.t.

$i < j$ and $A[i] > A[j]$

$(1, 2, 2, 2, 4, 0, 3, 3)$

Integer multiplication

(Reference: KT 5.5)
Problem: Given n-bit binary unsigned numbers x, y find $z = a * b$ $x * y$
where z has $2n$ bits.

Integer multiplication

(Reference: KT 5.5)

Problem: Given n-bit binary unsigned numbers x, y find $z = a * b$ where z has $2n$ bits.

Naive Method

```
   12            1100
 × 13          × 1101
 ----          ------
   36            1100
  12            0000
 ----          1100
  156         1100
              --------
              10011100
```

Integer multiplication

(Reference: KT 5.5)

Problem: Given n-bit binary unsigned numbers x, y find $z = a * b$ where z has $2n$ bits.

Naive Method

```
      12
    × 13
    ----
      36
     12
    ----
     156
```

```
        1100
      × 1101
    --------
        1100
       0000
      1100
     1100
    --------
    10011100
```

- $n - 1$ additions of n-bit numbers after shifting by i bits, $0 \leq i < n - 1$.

(Reference: KT 5.5)

Problem: Given n-bit binary unsigned numbers x, y find $z = a * b$ where z has $2n$ bits.

Naive Method

$$\begin{array}{r} 12 \\ \times\, 13 \\ \hline 36 \\ 12 \\ \hline 156 \end{array} \qquad \begin{array}{r} 1100 \\ \times\, 1101 \\ \hline 1100 \\ 0000 \\ 1100 \\ 1100 \\ \hline 10011100 \end{array}$$

- $n - 1$ additions of n-bit numbers after shifting by i bits, $0 \leq i < n - 1$.
- Each addition take $O(n)$ time.

Integer multiplication

(Reference: KT 5.5)

Problem: Given n-bit binary unsigned numbers x, y find $z = a * b$ where z has $2n$ bits.

Naive Method

$$\begin{array}{r} 12 \\ \times\, 13 \\ \hline 36 \\ 12 \\ \hline 156 \end{array} \qquad \begin{array}{r} 1100 \\ \times\, 1101 \\ \hline 1100 \\ 0000 \\ 1100 \\ 1100 \\ \hline 10011100 \end{array}$$

- $n-1$ additions of n-bit numbers after shifting by i bits, $0 \leq i < n-1$.
- Each addition take $O(n)$ time.
- Overall complexity $O(n^2)$.

A Divide and Conquer Solution

A Divide and Conquer Solution

- Split n-bit number x into two $n/2$-bit numbers x_1, x_2 in the middle.

A Divide and Conquer Solution

- Split n-bit number x into two $n/2$-bit numbers x_1, x_2 in the middle.
- $x = x_1 \cdot 2^{n/2} \; + \; x_0$ and $y = y_1 \cdot 2^{n/2} \; + \; y_0$.

A Divide and Conquer Solution

- Split n-bit number x into two $n/2$-bit numbers x_1, x_2 in the middle.
- $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.
- Hence,

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0. \end{aligned}$$

A Divide and Conquer Solution

- Split n-bit number x into two $n/2$-bit numbers x_1, x_2 in the middle.
- $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.
- Hence,

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0. \end{aligned}$$

- Complexity: $T(n)$

$T(1) = \Theta(1)$

$T(n) = 4 * T(n/2) + \Theta(n)$

A Divide and Conquer Solution

- Split n-bit number x into two $n/2$-bit numbers x_1, x_2 in the middle.
- $x = x_1 \cdot 2^{n/2} \ + \ x_0$ and $y = y_1 \cdot 2^{n/2} \ + \ y_0$.
- Hence,

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0. \end{aligned}$$

- Complexity: $T(n)$
  4 multiplications of $n/2$-bit numbers. $4 * T(n/2)$
  3 additions (with shifting) of $n/2$-bit numbers. $O(n)$.

A Divide and Conquer Solution

- Split n-bit number x into two $n/2$-bit numbers x_1, x_2 in the middle.
- $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.
- Hence,

$$xy = (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0)$$
$$= x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0.$$

$n^{\log_2(4)} = n^2$

- Complexity: $T(n)$
  4 multiplications of $n/2$-bit numbers. $4 * T(n/2)$
  3 additions (with shifting) of $n/2$-bit numbers. $O(n)$.
- Recurrence:

$$T(1) = c_1$$
$$T(n) = 4 * T(n/2) + c_2 * n \qquad \text{for } n > 1$$

A Divide and Conquer Solution

- Split n-bit number x into two $n/2$-bit numbers x_1, x_2 in the middle.
- $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.
- Hence,

$$xy = (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0)$$
$$= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0.$$

- Complexity: $T(n)$
  4 multiplications of $n/2$-bit numbers. $4 * T(n/2)$
  3 additions (with shifting) of $n/2$-bit numbers. $O(n)$.
- Recurrence:
  $T(1) = c_1$
  $T(n) = 4 * T(n/2) + c_2 * n$ for $n > 1$
- $T(n) = O(n^2)$. How?

Karatsuba Algorithm

- $xy = x_1 y_1 \cdot 2^n + (x_0 y_1 + x_1 y_0) \cdot 2^{n/2} + x_0 y_0$.

Karatsuba Algorithm

- $xy \;=\; x_1y_1 \cdot 2^n \;+\; (x_0y_1 + x_1y_0) \cdot 2^{n/2} \;+\; x_0y_0.$
- Terms x_1y_1, x_1y_0, x_0y_1, x_0y_0 are not independant.

- $xy = x_1y_1 \cdot 2^n + (x_0y_1 + x_1y_0) \cdot 2^{n/2} + x_0y_0$.
- Terms x_1y_1, x_1y_0, x_0y_1, x_0y_0 are not independant
- $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$

P

Karatsuba Algorithm

- $xy \;=\; x_1y_1 \cdot 2^n \;+\; (x_0y_1 + x_1y_0) \cdot 2^{n/2} \;+\; x_0y_0$.
- Terms x_1y_1, x_1y_0, x_0y_1, x_0y_0 are not independant.
- $(x_1 + x_0)(y_1 + y_0) \;=\; x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$
- Compute x_1y_1 and x_0y_0 and $p \;=\; (x_1 + x_0)(y_1 + y_0)$.

Karatsuba Algorithm

- $xy \;=\; x_1y_1 \cdot 2^n \;+\; (x_0y_1 + x_1y_0) \cdot 2^{n/2} \;+\; x_0y_0$.
- Terms x_1y_1, x_1y_0, x_0y_1, x_0y_0 are not independant.
- $(x_1 + x_0)(y_1 + y_0) \;=\; x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$
- Compute x_1y_1 and x_0y_0 and $p \;=\; (x_1 + x_0)(y_1 + y_0)$.
- Compute $(x_0y_1 + x_1y_0)$ as $p \;-\; (x_1y_1 + x_0y_0)$.

Karatsuba Algorithm

Recursive-Multiply(x,y):

  Write $x = x_1 \cdot 2^{n/2} + x_0$

    $y = y_1 \cdot 2^{n/2} + y_0$

  Compute $x_1 + x_0$ and $y_1 + y_0$

  p = Recursive-Multiply$(x_1 + x_0,\ y_1 + y_0)$

  $x_1 y_1$ = Recursive-Multiply(x_1, y_1)

  $x_0 y_0$ = Recursive-Multiply(x_0, y_0)

  Return $x_1 y_1 \cdot 2^n + (p - x_1 y_1 - x_0 y_0) \cdot 2^{n/2} + x_0 y_0$

$O(n)$

$\Theta(n)$

$T(n/2)$

$T(n/2)$

$T(n/2)$

Karatsuba Algorithm

```
Recursive-Multiply(x,y):
    Write x = x1 · 2^(n/2) + x0
          y = y1 · 2^(n/2) + y0
    Compute x1 + x0 and y1 + y0
    p = Recursive-Multiply(x1 + x0, y1 + y0)
    x1y1 = Recursive-Multiply(x1, y1)
    x0y0 = Recursive-Multiply(x0, y0)
    Return x1y1 · 2^n + (p − x1y1 − x0y0) · 2^(n/2) + x0y0
```

better $n \log(n) \cdot \log(\log(n))$

$\Rightarrow O(n^{1.59})$

Complexity

$$T(1) = c_1$$
$$T(n) = 3 * T(n/2) + c_2 * n \qquad \text{for } n > 1$$

#leaf = $n^{\log_2(3)} \cong n^{1.59}$

$f(n) = c_2 n$

Karatsuba Algorithm

```
Recursive-Multiply(x,y):
   Write x = x_1 · 2^{n/2} + x_0
         y = y_1 · 2^{n/2} + y_0
   Compute x_1 + x_0 and y_1 + y_0
   p = Recursive-Multiply(x_1 + x_0, y_1 + y_0)
   x_1y_1 = Recursive-Multiply(x_1, y_1)
   x_0y_0 = Recursive-Multiply(x_0, y_0)
   Return x_1y_1 · 2^n + (p − x_1y_1 − x_0y_0) · 2^{n/2} + x_0y_0
```

Complexity

$$
\begin{aligned}
T(1) &= c_1 \\
T(n) &= 3 * T(n/2) + c_2 * n \qquad \text{for} \quad n > 1
\end{aligned}
$$

Hence, $T(n) = O(n^{lg(3)}) = n^{1.59}$ (Why?)

Some Advanced Divide and Conquer Algorithms

- Strassen Matrix Multiplication. (ref: CLRS 4.2)
- Fast Fourier Transform. (ref: KT 5.6)
- Finding Closest pair of points in 2-D plane. (ref: KT 5.4)