بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر

(نیم‌سال تحصیلی ۴۰۲۲)

طراحی الگوریتم‌ها

حسین فلسفین

یک نکته دربارهٔ الگوریتم مبتنی بر تقسیم و غلبهٔ جلسهٔ پیش برای مسئلهٔ *Maxima-Set*

To analyze the divide-and-conquer maxima-set algorithm, there is a minor implementation detail in our algorithm that we need to work out. Namely, there is the issue of how to efficiently find the point, p, that is the median point in a lexicographical ordering of the points in S according to their (x, y)-coordinates.

There are two immediate possibilities:

☞ *One choice is to use a linear-time median-finding algorithm. This achieves a good asymptotic running time, but adds some implementation complexity.*

☞ *Another choice is to sort the points in S lexicographically by their (x, y)-coordinates as a preprocessing step, prior to calling the Maxima-Set algorithm on S. Given this preprocessing step, the median point is simply the point in the middle of the list. Moreover, each time we perform a recursive call, we can pass a sorted subset of S, which maintains the ability to easily find the median point each time.*

In either case, the rest of the nonrecursive steps can be performed in $O(n)$ time. The running time for the divide-and-conquer maxima-set algorithm is $O(n\log(n))$. (Why?)

Master Theorem

In the most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size $\frac{n}{2}$. More generally, an instance of size n can be divided into b instances of size $\frac{n}{b}$, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.) Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

where $f(n)$ is a function that accounts for the time spent on dividing an instance of size n into instances of size $\frac{n}{b}$ and combining their solutions. The recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ is called the general divide-and-conquer recurrence. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$.

The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem:

Master Theorem: ***If*** $f(n) \in \Theta(n^k)$ ***where*** $k \geq 0$ ***in the recurrence*** $T(n) = aT\left(\frac{n}{b}\right) + f(n)$***, then***

$$T(n) \in \begin{cases} \Theta(n^k), & \text{if } a < b^k, \\ \Theta(n^k \log(n)), & \text{if } a = b^k, \\ \Theta(n^{\log_b(a)}), & \text{if } a > b^k. \end{cases}$$

Of course, this approach can only establish a solution's order of growth to within an unknown multiplicative constant, whereas solving a recurrence equation with a specific initial condition yields an exact answer (at least for n*'s that are powers of* b*).*

Example:

$$\begin{array}{lll} & \begin{array}{ccc} a & b & k \\ \downarrow & \downarrow & \downarrow \end{array} & \\ T(n) = & 8\,T(n/4) + 5n^2 & \text{for } n > 1, n \text{ a power of } 4 \\ T(1) = 3 & & \end{array}$$

We have $8 < 4^2$**, therefore** $T(n) \in \Theta(n^2)$**.**

Example:

$$\begin{array}{lll} & \begin{array}{ccc} a & b & k \\ \downarrow & \downarrow & \downarrow \end{array} & \\ T(n) = & 9\,T(n/3) + 5n^1 & \text{for } n > 1, n \text{ a power of } 3 \\ T(1) = 7 & & \end{array}$$

We have $9 > 3^1$**, therefore** $T(n) \in \Theta(n^{\log_3(9)}) \in \Theta(n^2)$**.**

The maximum-subarray problem

We want to find the nonempty, contiguous subarray of A whose values have the largest sum. We call this contiguous subarray the maximum subarray. For example, in the following array, the maximum subarray of $A[1..16]$ is $A[8..11]$, with the sum 43:

	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

The brute-force solution takes $\Omega(n^2)$ time:

تعداد کل زیرآرایه‌هایی که باید در نظر گرفته شوند: $\binom{n}{2} + n$

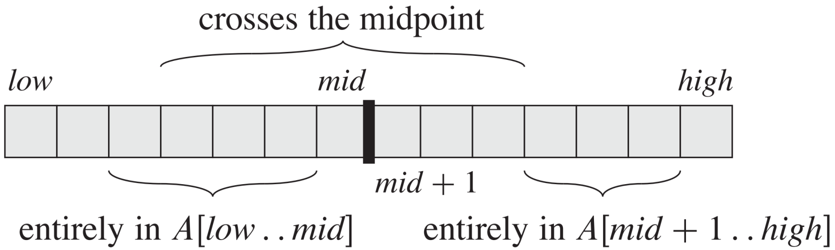
زمان لازم برای جمع عناصر یک زیرآرایه: $\Omega(1)$.

The maximum-subarray problem is interesting only when the array contains some negative numbers. If all the array entries were nonnegative, then the maximum-subarray problem would present no challenge, since the entire array would give the greatest sum.

Suppose we want to find a maximum subarray of the subarray $A[low..high]$. Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible. That is, we find the midpoint, say mid, of the subarray, and consider the subarrays $A[low..mid]$ and $A[mid..high]$.

Any contiguous subarray $A[i..j]$ of $A[low..high]$ must lie in exactly one of the following places:

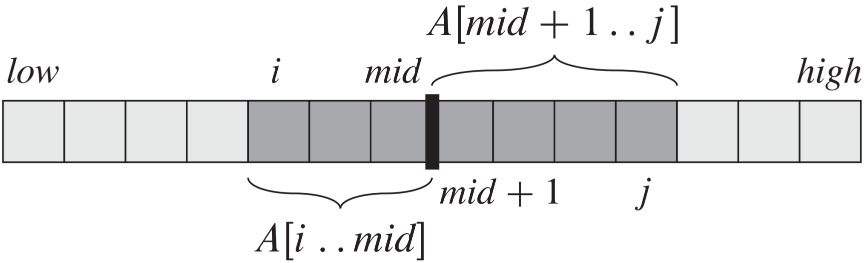
- ***Entirely in the subarray $A[low..mid]$, so that $low \leq i \leq j \leq mid$;***
- ***Entirely in the subarray $A[mid+1..high]$, so that $mid < i \leq j \leq high$; or***
- ***Crossing the midpoint, so that $low \leq i \leq mid < j \leq high$.***


crosses the midpoint
low
mid
high
mid + 1
entirely in A[low . . mid]
entirely in A[mid + 1 . . high]

We can find maximum subarrays of $A[low..mid]$ and $A[mid..high]$ recursively, because these two subproblems are smaller instances of the problem of finding a maximum subarray. Thus, all that is left to do is find a maximum subarray that crosses the midpoint, and take a subarray with the largest sum of the three.

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)
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)
```


A[mid + 1 . . j]
low
i
mid
high
mid + 1
j
A[i . . mid]

Any subarray crossing the midpoint is itself made of two subarrays $A[i..mid]$ and $A[mid+1..j]$, where $low \leq i \leq mid$ and $mid < j \leq high$. Therefore, we just need to find maximum subarrays of the form $A[i..mid]$ and $A[mid+1..j]$ and then combine them.

Lines 1–7 find a maximum subarray of the left half, $A[low..mid]$. Lines 8–14 work analogously for the right half, $A[mid+1..high]$. Finally, line 15 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($A, low, mid, high$) takes $\Theta(n)$ time. The for loop of lines 3–7 makes $mid - low + 1$ iterations, and the for loop of lines 10–14 makes $high - mid$ iterations, and so the total number of iterations is $(mid - low + 1) + (high - mid) = high - low + 1 = n$.

The initial call FIND-MAXIMUM-SUBARRAY($A, 1, A.length$) will find a maximum subarray of $A[1..n]$.

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

Lines 4 and 5 conquer by recursively finding maximum subarrays within the left and right subarrays, respectively.

Lines 6–11 form the combine part. Line 6 finds a maximum subarray that crosses the midpoint. (Recall that because line 6 solves a subproblem that is not a smaller instance of the original problem, we consider it to be in the combine part.) Line 7 tests whether the left subarray contains a subarray with the maximum sum, and line 8 returns that maximum subarray. Otherwise, line 9 tests whether the right subarray contains a subarray with the maximum sum, and line 10 returns that maximum subarray. If neither the left nor right subarrays contain a subarray achieving the maximum sum, then a maximum subarray must cross the midpoint, and line 11 returns it.

Analyzing the divide-and-conquer algorithm

We make the simplifying assumption that the original problem size is a power of 2, so that all subproblem sizes are integers. We denote by $T(n)$ the running time of FIND-MAXIMUM-SUBARRAY on a subarray of n elements.

Each of the subproblems solved in lines 4 and 5 is on a subarray of $\frac{n}{2}$ elements (our assumption that the original problem size is a power of 2 ensures that $\frac{n}{2}$ is an integer), and so we spend $T\left(\frac{n}{2}\right)$ time solving each of them. Because we have to solve two subproblems—for the left subarray and for the right subarray—the contribution to the running time from lines 4 and 5 comes to $2T\left(\frac{n}{2}\right)$. The call to FIND-MAX-CROSSING-SUBARRAY in line 6 takes $\Theta(n)$ time.

Master method: *This recurrence has the solution* $T(n) \in \Theta(n\log(n))$.

We see that the divide-and-conquer method yields an algorithm that is asymptotically faster than the brute-force method. There is in fact a linear-time algorithm for the maximum-subarray problem, and it does not use divide-and-conquer.

The Closest-Pair Problem

Let P be a set of $n > 1$ points in the Cartesian plane. For the sake of simplicity, we assume that the points are distinct. We can also assume that the points are ordered in nondecreasing order of their x coordinate. (If they were not, we could sort them first by an efficeint sorting algorithm such as mergesort.) It will also be convenient to have the points sorted in a separate list in nondecreasing order of the y coordinate; we will denote such a list Q.

We assume that the points in question are specified in a standard fashion by their (x, y) Cartesian coordinates and that the distance between two points $p_i(x_i, y_i)$ and $p_j(x_j, y_j)$ is the standard Euclidean distance

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

ALGORITHM *BruteForceClosestPair*(P)

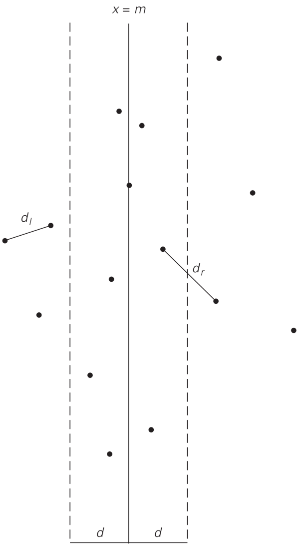
//Finds distance between two closest points in the plane by brute force
//Input: A list P of n $(n \geq 2)$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$
//Output: The distance between the closest pair of points
$d \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //*sqrt* is square root
return d

$T(n) \in \Theta(n^2)$ ***(why?)***

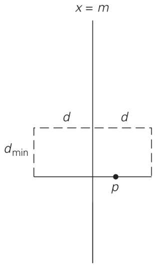
If $2 \leq n \leq 3$, the problem can be solved by the obvious brute-force algorithm. If $n > 3$, we can divide the points into two subsets P_l and P_r of $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$ points, respectively, by drawing a vertical line through the median m of their x coordinates so that $\lceil \frac{n}{2} \rceil$ points lie to the left of or on the line itself, and $\lfloor \frac{n}{2} \rfloor$ points lie to the right of or on the line. Then we can solve the closest-pair problem recursively for subsets P_l and P_r. Let d_l and d_r be the smallest distances between pairs of points in P_l and P_r, respectively, and let $d = \min\{d_l, d_r\}$.

As in most divide-and-conquer algorithms, most of the work comes from the combine step:

The points of a closer pair can lie on the opposite sides of the separating line. As a step combining the solutions to the smaller subproblems, we need to examine such points. Obviously, we can limit our attention to the points inside the symmetric vertical strip of width $2d$ around the separating line, since the distance between any other pair of points is at least d.



(a)



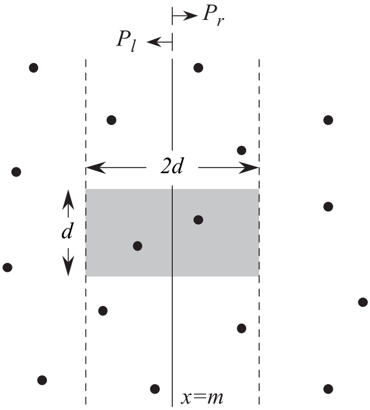
(b)

Let S be the list of points inside the strip of width $2d$ around the separating line, obtained from Q and hence ordered in non-decreasing order of their y coordinate. We will scan this list, updating the information about $d_{\min}$, the minimum distance seen so far, if we encounter a closer pair of points. Initially, $d_{\min} = d$, and subsequently $d_{\min} \leq d$.

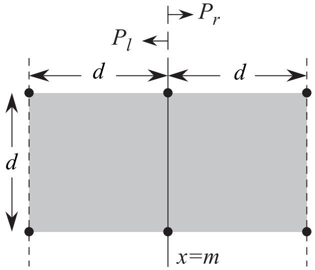
Let $p(x, y)$ be a point on this list. For a point $p(x, y)$ to have a chance to be closer to p than $d_{\min}$, the point must follow p on list S and the difference between their y coordinates must be less than $d_{\min}$ (why?). Geometrically, this means that p must belong to the rectangle shown in the above figure.

The points lying within the two strips of width d around the separating line have a special structure.

The principal insight exploited by the algorithm is the observation that the rectangle can contain just a few such points, because the points in each half (left and right) of the rectangle must be at least distance d apart. It is easy to prove that the total number of such points in the rectangle, including p, does not exceed six. Thus, the algorithm can consider no more than five next points following p on the list S, before moving up to the next point.



(a)



(b)

نکته: اگر فرض متمایز بودن نقاط را برداریم، آنگاه مستطیل $d \times 2d$ ما می‌تواند حداکثر حاوی ۸ نقطه باشد:

If the distance between any two points in the $d \times 2d$ rectangle must be at most d, then the rectangle can accommodate at most eight points: at most four points from P_l and at most four points from P_r. The maximum number is attained when one point from P_l coincides with one point from P_r at the intersection of the vertical line with the top of the rectangle, and one point from P_l coincides with one point from P_r at the intersection of the vertical line with the bottom of the rectangle.

But we assumed that there is no coincidence.
Observation: Each point in the strip needs to be compared with at most five points.

```
ALGORITHM  EfficientClosestPair(P, Q)
    //Solves the closest-pair problem by divide-and-conquer
    //Input: An array P of n ≥ 2 points in the Cartesian plane sorted in
    //       nondecreasing order of their x coordinates and an array Q of the
    //       same points sorted in nondecreasing order of the y coordinates
    //Output: Euclidean distance between the closest pair of points
    if n ≤ 3
        return the minimal distance found by the brute-force algorithm
    else
        copy the first ⌈n/2⌉ points of P to array P_l
        copy the same ⌈n/2⌉ points from Q to array Q_l
        copy the remaining ⌊n/2⌋ points of P to array P_r
        copy the same ⌊n/2⌋ points from Q to array Q_r
        d_l ← EfficientClosestPair(P_l, Q_l)
        d_r ← EfficientClosestPair(P_r, Q_r)
        d ←min{d_l, d_r}
        m ← P[⌈n/2⌉ − 1].x
        copy all the points of Q for which |x − m| < d into array S[0..num − 1]
        dminsq ← d^2
        for i ← 0 to num − 2 do
            k ← i + 1
            while k ≤ num − 1 and (S[k].y − S[i].y)^2 < dminsq
                dminsq ← min((S[k].x − S[i].x)^2+ (S[k].y − S[i].y)^2, dminsq)
                k ← k + 1
    return sqrt(dminsq)
```

The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions. Therefore, assuming as usual that n is a power of 2, we have the following recurrence for the running time of the algorithm:

$$T(n) = 2T\left(n/2\right) + f(n),$$

where $f(n) \in \Theta(n)$.
Master Theorem: $T(n) \in \Theta(n \log n)$.

The necessity to presort input points does not change the overall efficiency class if sorting is done by a $O(n \log n)$ algorithm such as mergesort. In fact, this is the best efficiency class one can achieve, because it has been proved that any algorithm for this problem must be in $\Theta(n \log n)$ under some natural assumptions about operations an algorithm can perform.