

Divide and Conquer

Tao Hou

The Divide-and-Conquer Paradigm

- **Divide phase:** Divide the problem into subproblems
- **Conquer phase:** Conquer/solve the subproblems (recursively)
- **Combine phase:** Combine the solutions to the subproblems into a solution for the whole problem

Example: Merge Sort (Review)

- **Divide phase:** Divide the array into two halves from the middle
- **Conquer Phase:** Sort each half recursively
- **Combine phase:** Merge the two sorted halves

```
MERGESORT(A)
1  if  $length(A) == 1$ 
2      return A
3   $m = \lfloor length(A)/2 \rfloor$ 
4   $A_L = \textbf{MERGESORT}(A[1..m])$ 
5   $A_R = \textbf{MERGESORT}(A[m+1..length(A)])$ 
6  return MERGE( $A_L, A_R$ )
```

- What the MERGE routine does: given two sorted arrays, return a single sorted array containing all elements of the given two arrays
- The MERGE routine runs in $O(n)$ time where n is the size of the larger given array

MERGE(A, B)

```
1   $i, j = 1$ 
2   $X = \emptyset$ 
3  while  $i \leq \text{length}(A)$  and  $j \leq \text{length}(B)$ 
4      if  $A[i] \leq B[j]$ 
5           $X = X \circ A[i]$            // appends  $A[i]$  to  $X$ 
6           $i = i + 1$ 
7      else
8           $X = X \circ B[j]$ 
9           $j = j + 1$ 
10 while  $i \leq \text{length}(A)$ 
11      $X = X \circ A[i]$ 
12      $i = i + 1$ 
13 while  $j \leq \text{length}(B)$ 
14      $X = X \circ B[j]$ 
15      $j = j + 1$ 
16 return  $X$ 
```

Run-Time Analysis of Merge Sort

Input Size: n

$$T(n) = \begin{cases} C_1 & \text{if } n=1 \\ 2T(n/2) + n * C_2 & \text{otherwise} \end{cases}$$

Q: How to solve it?

The Master Theorem

Let $a \geq 1$, $b > 1$, $f(n) = O(n^d)$ where $d \geq 0$, and $c = \log_b a$

$$T(n) = \begin{cases} O(1) & \text{if } n = O(1) \\ aT(n/b) + f(n) & \text{otherwise} \end{cases}$$

1. $c < d$: $T(n) = \Theta(f(n)) = \Theta(n^d)$
2. $c > d$: $T(n) = \Theta(n^c)$
3. $c = d$: $T(n) = \Theta(n^c \log n)$

The Master Theorem

Let $a \geq 1$, $b > 1$, $f(n) = O(n^d)$ where $d \geq 0$, and $c = \log_b a$

$$T(n) = \begin{cases} O(1) & \text{if } n = O(1) \\ aT(n/b) + f(n) & \text{otherwise} \end{cases}$$

1. $c < d$: $T(n) = \Theta(f(n)) = \Theta(n^d)$
2. $c > d$: $T(n) = \Theta(n^c)$
3. $c = d$: $T(n) = \Theta(n^c \log n)$

Remark: For case 1, $f(n)$ must also satisfy a *regularity condition* which states that there is some $C < 1$ such that $a \cdot f(n/b) \leq C \cdot f(n)$ for sufficiently large n . This regularity condition is almost always true and we will not worry about it.

Run-Time Analysis of Merge Sort using Master Theorem

$$T(n) = \begin{cases} C_1 & \text{if } n=1 \\ 2T(n/2) + C_2 \cdot n & \text{otherwise} \end{cases}$$

Applying the Master Theorem with $a = 2$, $b = 2$, and $d = 1$, we get $c = \log_2 2 = d$ and $T(n) = \Theta(n \log n)$

Master Theorem: Additional Examples

Example 1

$$T(n) = T(n/2) + 5n$$

Applying the Master Theorem with $a = 1$, $b = 2$, $d = 1$, we get $c = 0 < d$ and hence $T(n) = \Theta(n)$

Master Theorem: Additional Examples

Example 1

$$T(n) = T(n/2) + 5n$$

Applying the Master Theorem with $a = 1$, $b = 2$, $d = 1$, we get $c = 0 < d$ and hence $T(n) = \Theta(n)$

Example 2

$$T(n) = 4T(n/2) + 2n$$

Applying the Master Theorem with $a = 4$, $b = 2$, $d = 1$, we get $c = 2 > d$ and hence $T(n) = \Theta(n^2)$

Examples: Using the Master Theorem

Example 3

$$T(n) = T(n - 5) + n$$

- The Master Theorem does not apply here.
- The *iteration method* (briefly reviewed next) can be used to solve this equation

Run-Time Analysis of Merge Sort (Iteration Method)

We can also solve $T(n)$ using the *Iteration Method* (aka. keep on expanding the formula by applying $T(n)$ to itself, until reaching the base case):

$$(1) : T(n) = 2T(n/2) + C_2n$$

$$(2) : T(n) = 2^2 T(n/2^2) + 2C_2n$$

$$(3) : T(n) = 2^3 T(n/2^3) + 3C_2n$$

$$\vdots$$

$$(i) : T(n) = 2^i T(n/2^i) + i \cdot C_2n$$

We stop iterating when $n/2^i = 1$

Setting $n/2^i = 1$ gives a number of iterations $i = \log n$

Plugging the value of $i = \log n$ gives:

$$T(n) = 2^i T(n/2^i) + i \cdot C_2n = 2^{\log n} C_1 + n \cdot \log n = nC_1 + \log n \cdot C_2 \cdot n = \Theta(n \log n)$$

- **Divide:** Partition A into $A[1 \dots q - 1]$ and $A[q + 1 \dots n]$ such that

$$A[1], \dots, A[q - 1] \leq A[q] \leq A[q + 1], \dots, A[n]$$

- ▶ The partition is done by a **PARTITION** procedure which may change the positions of elements
 - ▶ q is returned from the partition procedure and in general we don't have any control over q
- **Conquer:** Sort $A[1 \dots q - 1]$ and $A[q + 1 \dots n]$ recursively
- **Combine:** Nothing to do here

```
QUICKSORT( $A, begin, end$ )
```

```
1  if  $begin < end$   
2       $q = \mathbf{PARTITION}(A, begin, end)$   
3      QUICKSORT( $A, begin, q - 1$ )  
4      QUICKSORT( $A, q + 1, end$ )
```

```
PARTITION(A, begin, end)  
1  q = begin  
2  v = A[end]  
3  for i = begin to end - 1  
4      if A[i] < v  
5          swap A[i] and A[q]  
6          q = q + 1  
7  swap A[q] and A[end]  
8  return q
```

■ Runs in $\Theta(n)$ time

■ Further remarks:

- ▶ Assume a *pivot* (center of the partition) *v* to be at the end
- ▶ Loop invariant (always **true** at the beginning of each iteration):
q is a separation of *A*[*begin* . . . *i* - 1] s.t.

$$A[\textit{begin}], \dots, A[q - 1] < v \text{ and } A[q], \dots, A[i - 1] \geq v$$

Worst-Case Run-Time Analysis of Quick Sort (Review)

Input Size: n

Worst Case: The array partition is very skewed: 0 element on one side, pivot, and the rest on the other side (the pivot is the smallest or largest element)

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n-1) + n & \text{otherwise} \end{cases}$$

We cannot solve $T(n)$ using the master method.

Using the Iteration Method

We solve $T(n)$ by expanding the recursive formula directly:

$$(1) : T(n) = T(n-1) + n$$

$$(2) : T(n) = T(n-2) + n - 1 + n$$

$$(3) : T(n) = T(n-3) + n - 2 + n - 1 + n$$

.

.

$$(i) : T(n) = T(n-i) + (n-i+1) + (n-i+2) + \cdots + n$$

We stop expanding when $n-i=1$

Setting $n-i=1$ gives $i=n-1$

Plugging this value of i in the generic form gives

$$T(n) = T(1) + 2 + 3 + \cdots + n = 1 + 2 + 3 + \cdots + n = n(n+1)/2 = \Theta(n^2)$$

Average-Case Run-Time Analysis of Quick Sort (Advanced)

- Idea: count the number of comparisons
- Rename elements (assumed to be distinct) in A as $z_1 < z_2 < \dots < z_n$
- Define a random variable X_{ij} as:

$$X_{ij} = \begin{cases} 0 & \text{if } z_i \text{ and } z_j \text{ does not compare} \\ 1 & \text{if } z_i \text{ and } z_j \text{ does compare} \end{cases}$$

- The random variable for the number of comparison is:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}$$

Average-Case Run-Time Analysis of Quick Sort (Advanced)

- We have

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] \end{aligned}$$

- By some analysis (we omit),

$$E[X_{ij}] = \frac{2}{j-i+1}$$

Average-Case Run-Time Analysis of Quick Sort (Advanced)

- Then

$$\begin{aligned} E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} \\ &< \sum_{i=1}^{n-1} \left(2 \sum_{k=1}^{\infty} \frac{1}{k} \right) \quad (\text{inner sum } \textit{harmonic series}) \\ &= \sum_{i=1}^{n-1} O(\log n) = O(n \log n) \end{aligned}$$

Selection

Problem

Given an (unsorted) array $A[1 \dots n]$ of numbers and $k \in \mathbb{N}$, find the k -th smallest number in A

A First Random Solution

- (i) **Divide:** Randomly select a pivot from A , partition A into two subarrays L and R s.t. elements in $L \leq$ elements in R
- (ii) **Conquer:** If $k \leq |L|$, recurse to find the k -th smallest element in L ; otherwise, recurse to find the $(k - |L|)$ -th smallest element in R

Random select

RandSelect(A, k)

1. **if** $|A| == 1$ **then return** $A[1]$;
2. $L, R = \text{Partition}(A)$;
3. **if** $k \leq |L|$ **then return** $\text{RandSelect}(L, k)$;
4. **else return** $\text{RandSelect}(R, k - |L|)$;

Figure 1: The algorithm RandSelect

Random solution: Complexity

(Analysis similar to quicksort)

- Best case:

Random solution: Complexity

(Analysis similar to quicksort)

- Best case: $O(n)$

Random solution: Complexity

(Analysis similar to quicksort)

- Best case: $O(n)$
- Worst case:

Random solution: Complexity

(Analysis similar to quicksort)

- Best case: $O(n)$
- Worst case: $O(n^2)$

Random solution: Complexity

(Analysis similar to quicksort)

- Best case: $O(n)$
- Worst case: $O(n^2)$
- Average case:

Random solution: Complexity

(Analysis similar to quicksort)

- Best case: $O(n)$
- Worst case: $O(n^2)$
- Average case: $O(n)$

Linear-Time Selection

Can we have a selection algorithm which runs in linear time in the worst case?

- Observe that the previous random selection runs in quadratic time because sometimes the partition can be unbalanced
- Can we try to choose a “good” pivot for the partition each time so that the partitioned arrays are always balanced?

Linear-Time Selection

Can we have a selection algorithm which runs in linear time in the worst case?

- Observe that the previous random selection runs in quadratic time because sometimes the partition can be unbalanced
- Can we try to choose a “good” pivot for the partition each time so that the partitioned arrays are always balanced?
- The answer is that we can

Linear-Time Selection

Solution:

- (i) Partition the array into $m = \lceil n/5 \rceil$ subarrays, each consisting of 5 (maybe less) consecutive elements
- (ii) Find the median of each of the m arrays by brute force
- (iii) Recursively find the median M of the m medians
- (iv) Using M as pivot, partition A into two subarrays L and R
- (v) If $k \leq |L|$, recurse to find the k -th smallest element in L ; otherwise, recurse to find the $(k - |L|)$ -th smallest element in R

The Selection Algorithm

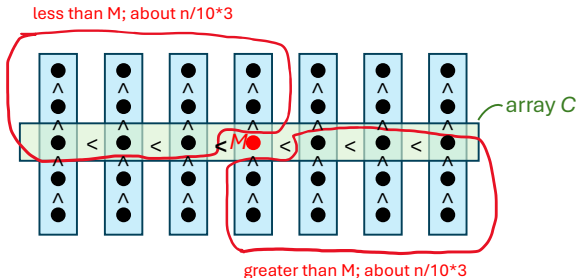
Select(A, k)

1. **if** $n \leq 25$ **then return** the k -th smallest element in A by brute force;
2. $m = \lceil n/5 \rceil$; create an array $C[1..m]$;
3. **for** $i = 1$ **to** m $C[i] :=$ the median of $A[(5i - 4)..(5i)]$;
4. $M = \text{Select}(C, m/2)$;
5. Partition A using M as the pivot into L and R , where L contains all elements that are smaller or equal to M and R contains the rest;
6. **if** $k \leq |L|$ **then return** $\text{Select}(L, k)$;
7. **else return** $\text{Select}(R, k - |L|)$;

Figure 2: The algorithm Select

Run-Time Analysis of Select

- Take $n = 35$
- For simplicity, assume all elements are distinct
- Order each small array, and then order the 7 small arrays by their medians



Run-Time Analysis of Select

In general:

- Ignore the floors and ceilings
- The number of medians in the array C less than M is:
 $(1/2) \cdot (n/5) = n/10$
- The number of other elements less than M is at least: $2n/10$
- So, at least $3n/10$ elements is less than M
- Similarly, at least $3n/10$ elements is greater than M
- Whether we go to L or R in the algorithm, we drop at least $3n/10$ elements (i.e., keep at most $7n/10$ elements).

Run-Time Analysis of Select

$$T(n) \leq \begin{cases} O(1) & \text{if } n \leq 25 \\ T(7n/10) + T(n/5) + O(n) & \text{otherwise} \end{cases}$$

We cannot solve $T(n)$ using the master method.

Instead, use the *substitution* method:

- 1 Guess the solution
- 2 Plug in the guess and prove the equation to be true based on the assumption that the equation is true for sub-cases

Notice: The substitution method is in some sense a proof by *induction*

Run-Time Analysis of Select

- Our induction hypothesis:
 - Suppose that $T(i) \leq c \cdot i$ for any $i < n$, where c is a constant
 - Want to prove that $T(n) \leq c \cdot n$, i.e., $T(n) = O(n)$

Run-Time Analysis of Select

- Our induction hypothesis:
 - Suppose that $T(i) \leq c \cdot i$ for any $i < n$, where c is a constant
 - Want to prove that $T(n) \leq c \cdot n$, i.e., $T(n) = O(n)$
- We have

$$\begin{aligned}T(n) &\leq T(7n/10) + T(n/5) + O(n) \\&\leq c \cdot (7n/10) + c \cdot (n/5) + c'n \\&= 9cn/10 + c'n = cn \cdot (9/10 + c'/c)\end{aligned}$$

Run-Time Analysis of Select

- Our induction hypothesis:
 - Suppose that $T(i) \leq c \cdot i$ for any $i < n$, where c is a constant
 - Want to prove that $T(n) \leq c \cdot n$, i.e., $T(n) = O(n)$
- We have

$$\begin{aligned}T(n) &\leq T(7n/10) + T(n/5) + O(n) \\&\leq c \cdot (7n/10) + c \cdot (n/5) + c'n \\&= 9cn/10 + c'n = cn \cdot (9/10 + c'/c)\end{aligned}$$

- So we only need to choose a c s.t. $c'/c + 9/10 \leq 1$, which is $c \geq 10c'$, so that we will have

$$T(n) \leq cn \cdot (9/10 + c'/c) \leq cn$$

The Closest Pair of Points

Problem

Given a set $S = \{p_1, \dots, p_n\}$ of points in the plane, where $p_i = (x_i, y_i)$, compute a closest-pair of points in S , that is, a pair of distinct points $p_i, p_j \in S$ such that $|p_i p_j| = \min\{|p_r p_s| : p_r \neq p_s \in S\}$

Note: we assume the points in S to have *distinct* coordinates; if there are duplicate points in S , this is easy to pre-check and the answer is 0

The Closest-Pair Algorithm: Overview

- **Divide:** Partition the input set S into two sets S_L and S_R of the same size s.t. points in S_L are to the left of points in S_R
- **Conquer:** Recursively find the minimum distances of S_L and S_R
- **Combine:** Find the minimum distance of point pairs where one is from S_L and the other is from S_R ; return the minimum of the three minimums

The Closest-Pair Algorithm: Overview

- **Divide:** Partition the input set S into two sets S_L and S_R of the same size s.t. points in S_L are to the left of points in S_R
- **Conquer:** Recursively find the minimum distances of S_L and S_R
- **Combine:** Find the minimum distance of point pairs where one is from S_L and the other is from S_R ; return the minimum of the three minimums

We aim to achieve $O(n)$ time for both the divide and combine phase so that the entire complexity is $O(n \log n)$

Preprocessing Step

- Let X be a list containing the points in S sorted w.r.t. their x -coordinates, and Y a list containing the points in S sorted w.r.t. their y -coordinates. Clearly, X and Y can be obtained in $O(n \log n)$ time (we only do this *once* at the beginning).

So the input to the algorithm, i.e., the set of points, is encoded as a tuple of three arrays (S, X, Y)

Divide Phase

- Partition S into S_L and S_R of equal size s.t. points in S_L are to the *left* of S_R using a central vertical line D
- Let X_L , Y_L each represent the set of points in S_L sorted by x- and y-coordinates respectively; X_R and Y_R are similarly defined for S_R

Divide Phase: Pseudocode

1. $m = |X|/2$
2. $D = X[m].x$
3. $X_L = X[1 \dots m]$
4. $X_R = X[m + 1 \dots |X|]$
5. **for** $i = 1 \dots |Y|$:
6. **if** $Y[i].x \leq D$:
7. append $Y[i]$ to Y_L
8. **else**:
9. append $Y[i]$ to Y_R
10. separate S into S_L, S_R similarly

Figure 3: Divide phase

Conquer Phase

- Recursively call the algorithm on (S_L, X_L, Y_L) to obtain the min-distance δ_L for S_L , and on (S_R, X_R, Y_R) to obtain the min-distance δ_R for S_R .

Combine Phase

Idea

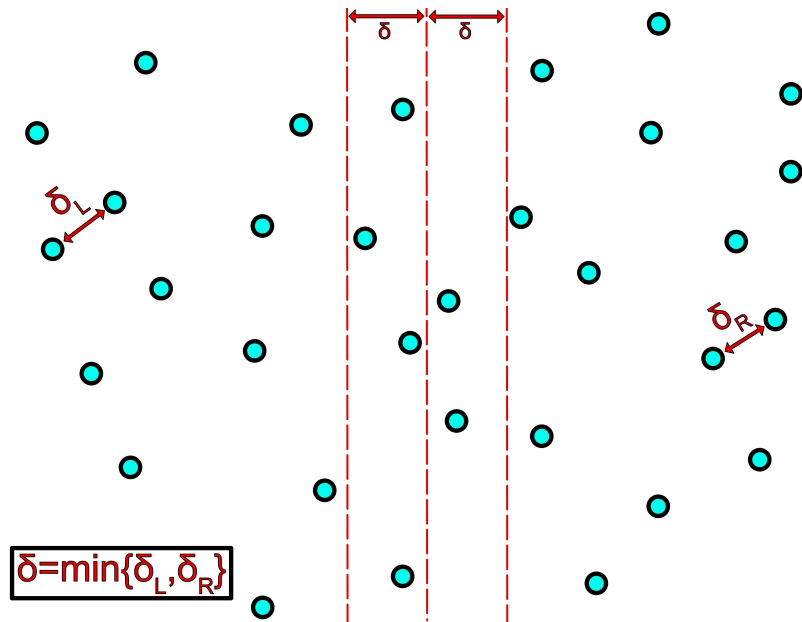
- We have:
 - δ_L : The min dis of pairs in S_L
 - δ_R : The min dis of pairs in S_R
- Aim of combine phase: Compute the min-dis of the pairs where one point is from S_L and the other is from S_R (i.e., pairs of points from different sides)
- Answer: The minimum of above three minimums

Details of Combine Phase

The first observation

- Let $\delta = \min\{\delta_L, \delta_R\}$
- We only need to consider pairs within a 2δ -**wide vertical strip centered around D**

Consider only 2δ -wide vertical strip centered around D



Explanation

- We have computed min-dis of points from the same side, which is δ .
- So, to compute the overall min-dis, we can ignore those point pairs whose distances are greater than δ .
- If two points from different sides are not both from the 2δ -wide vertical strip (at least one point is outside the strip), then their distance is greater than δ , and so we can ignore them.

The Combine Phase

- Let $\delta = \min\{\delta_L, \delta_R\}$
- From Y , create Y_{mid} (also sorted by y-coordinates) which is the set of points within the 2δ -wide vertical strip centered around D

The Combine Phase

- Let $\delta = \min\{\delta_L, \delta_R\}$
- From Y , create Y_{mid} (also sorted by y-coordinates) which is the set of points within the 2δ -wide vertical strip centered around D
- Go over Y_{mid} , and for each point p , compute its distance to **at most 7** points in Y_{mid} that follow p , and keep track of the min-distance
- Return the smaller of δ and what we have by scanning Y_{mid}

Combine Phase: Pseudocode

```
1. for  $i = 1 \dots |Y|$ :  
2.   if  $Y[i].x \geq D - \delta$  and  $Y[i].x \leq D + \delta$ :  
3.     append  $Y[i]$  to  $Y_{mid}$   
4.  $\bar{\delta} = \infty$   
5. for  $i = 1 \dots |Y_{mid}|$ :  
6.   for  $j = 1 \dots 7$ :  
7.     if  $i + j \leq |Y_{mid}|$  and  $\text{dis}(Y_{mid}[i], Y_{mid}[i + j]) < \bar{\delta}$  then  
8.        $\bar{\delta} = \text{dis}(Y_{mid}[i], Y_{mid}[i + j])$   
9. return  $\min\{\delta, \bar{\delta}\}$ 
```

Figure 4: Combine phase

Why only scan 7 points?

- For each point p in Y_{mid} , we only need to consider other points in Y_{mid} whose distances to p is $< \delta$. This means we only need to consider points **within a $2\delta \times 2\delta$ square** of p .

Why only scan 7 points?

- For each point p in Y_{mid} , we only need to consider other points in Y_{mid} whose distances to p is $< \delta$. This means we only need to consider points **within a $2\delta \times 2\delta$ square** of p .
- **Key observation:** Each $\delta \times \delta$ square contains **at most 4 points**
 - This square is totally within the left or right side of the vertical separator D , meaning that points in the square are either all from S_L or all from S_R , so these points are at least δ -distance apart
 - A fact from computational geometry says that such a square cannot fit in more than 4 points

Why only scan 7 points?

- Therefore, each $2\delta \times \delta$ square contains at most 8 points (including p)
- So we only need to scan the 7 points that precede p (ones that are in the upper $2\delta \times \delta$ square) and the 7 points that follow p (ones that are in the lower $2\delta \times \delta$ square) in Y_{mid} .
- Further observation: we only need to scan the 7 points that follow p , and ignore the 7 points that precede p :
 - Suppose there is a point q preceding p in Y_{mid} falling within the upper $2\delta \times \delta$ square for p . Then p also falls in the lower $2\delta \times \delta$ square for q . So we have checked the pair p, q when we scan q .

The Closest-Pair Algorithm

Closest-Pair-Algo

1. **if** $|S| \leq 3$ **return** a closest pair (p_{min}, q_{min}) in S by brute force;
2. using X , compute a vertical line D of equation $x = \ell$ that partitions S into S_L, S_R of equal size such that all points in S_L are on D or to the left of it, and all points in S_R are on D or to the right of it;
3. using X and Y , create the arrays X_L, Y_L and X_R, Y_R ;
4. recurse on S_L, X_L, Y_L to compute a closest pair (p_L, q_L) ; let $\delta_L = |p_L q_L|$;
5. recurse on S_R, X_R, Y_R to compute a closest pair (p_R, q_R) ; let $\delta_R = |p_R q_R|$;
6. let $\delta = \min \{\delta_L, \delta_R\}$;
7. let S_{mid} be the set of points in S whose x -coordinate satisfies $\ell - \delta \leq x \leq \ell + \delta$;
8. using Y , compute the list of points in S_{mid} sorted by their y -coordinates;
9. go over Y_{mid} (in the sorted order), and for each point, compute its distance to the next (at most) 7 points in Y_{mid} and keep track of the pair of points (p_{mid}, q_{mid}) of minimum distance;
10. return the closest pair (p_{min}, q_{min}) among, (p_L, q_L) , (p_R, q_R) , and (p_{mid}, q_{mid}) ;

Figure 5: The algorithm Closest Pair

Run-Time Analysis of Closest-Pair

Let $T(n)$ be the running time of Closest-Pair in the worst case on n points.

- Divide phase takes $O(n)$ time.
- Combine phase takes $O(n)$ time.
- Recursive call on (S_L, X_L, Y_L) takes $T(n/2)$ time;
recursive call on (S_R, X_R, Y_R) takes $T(n/2)$ time.

Therefore, $T(n)$ obeys the following recurrence relation:

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

We can solve $T(n)$ using the Master Theorem to obtain $T(n) = O(n \lg n)$

Integer Multiplication

Problem

Multiply two integers x, y represented as sequences (e.g., arrays) of 0-1 bits where the lengths of the sequences can be **arbitrarily** large (assume the length of the two to be both n , with possibly padding 0's)

Notice: This cannot be simply done in constant time: the multiplication of provided by the CPU only supports a *fixed* length on the sequence (e.g., 64).

An Algorithm Everybody Knows

Solution:

- Compute a “partial product” by multiplying each digit of y separately by x , and then you add up all the partial products.
- Only this time we do the *binary* version, i.e., we multiplying each *bit* of y by x and then add up.

	1100
	$\times 1101$
	<hr/>
	1100
	0000
	1100
	1100
	<hr/>
	10011100
(a)	(b)

(Figure taken from [Kleinberg&Tardos - Algorithm Design])

Time complexity:

An Algorithm Everybody Knows

Solution:

- Compute a “partial product” by multiplying each digit of y separately by x , and then you add up all the partial products.
- Only this time we do the *binary* version, i.e., we multiplying each *bit* of y by x and then add up.

	1100
	$\times 1101$
	<hr/>
	1100
	0000
	1100
	<hr/>
	1100
	10011100
(a)	(b)

$$\begin{array}{r} 12 \\ \times 13 \\ \hline 36 \\ 12 \\ \hline 156 \end{array}$$

(Figure taken from [Kleinberg&Tardos - Algorithm Design])

Time complexity: $O(n^2)$

Attempting to Improve the Naive Algorithm

Using, of course, divide and conquer:

Attempting to Improve the Naive Algorithm

Using, of course, divide and conquer:

- Write x as $x = x_1 \cdot 2^{n/2} + x_0$, where x_1 is the “high-order” half bits x_0 is the “low-order” half bits
- Similarly write y as $y = y_1 \cdot 2^{n/2} + y_0$

Attempting to Improve the Naive Algorithm

Using, of course, divide and conquer:

- Write x as $x = x_1 \cdot 2^{n/2} + x_0$, where x_1 is the “high-order” half bits x_0 is the “low-order” half bits
- Similarly write y as $y = y_1 \cdot 2^{n/2} + y_0$
- Rewrite xy as

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

Attempting to Improve the Naive Algorithm

Using, of course, divide and conquer:

- Write x as $x = x_1 \cdot 2^{n/2} + x_0$, where x_1 is the “high-order” half bits x_0 is the “low-order” half bits
- Similarly write y as $y = y_1 \cdot 2^{n/2} + y_0$
- Rewrite xy as

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

So, to compute xy (multiplying two n -sequences), we only need to:

- Recursively compute four multiplications of $n/2$ -sequences:

$$x_1y_1, x_1y_0, x_0y_1, \text{ and } x_0y_0$$

- Then take the sum $x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$ (which can be done in $O(n)$ time)

Recursive-Multiply(x, y)

1. write $x = x_1 \cdot 2^{n/2} + x_0$, $y = y_1 \cdot 2^{n/2} + y_0$
2. $x_1y_1 = \text{Recursive-Multiply}(x_1, y_1)$
3. $x_1y_0 = \text{Recursive-Multiply}(x_1, y_0)$
4. $x_0y_1 = \text{Recursive-Multiply}(x_0, y_1)$
5. $x_0y_0 = \text{Recursive-Multiply}(x_0, y_0)$
6. **return** $x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$

Figure 6: Recursive Multiply

Time complexity:

Recursive-Multiply(x, y)

1. write $x = x_1 \cdot 2^{n/2} + x_0$, $y = y_1 \cdot 2^{n/2} + y_0$
2. $x_1y_1 = \text{Recursive-Multiply}(x_1, y_1)$
3. $x_1y_0 = \text{Recursive-Multiply}(x_1, y_0)$
4. $x_0y_1 = \text{Recursive-Multiply}(x_0, y_1)$
5. $x_0y_0 = \text{Recursive-Multiply}(x_0, y_0)$
6. **return** $x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$

Figure 6: Recursive Multiply

Time complexity:

- $T(n) = 4T(n/2) + O(n)$ which is $O(n^2)$ (no improvement at all!)

Second Attempt to Improve the Naive Algorithm

- The problem with the previous divide-and-conquer approach is that it involves **four** recursive calls
- If we can reduce the number of recursive calls to **three**, we would have

$$T(n) = 3T(n/2) + O(n)$$

which is $O(n^{1.59})$ (quite an improvement!)

Second Attempt to Improve the Naive Algorithm

- Notice that our goal is to compute the sum

$$xy = x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0 \quad (1)$$

Second Attempt to Improve the Naive Algorithm

- Notice that our goal is to compute the sum

$$xy = x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0 \quad (1)$$

- Consider another multiplication

$$p = (x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$$

where we observe $x_1y_0 + x_0y_1 = p - x_1y_1 - x_0y_0$

Second Attempt to Improve the Naive Algorithm

- Notice that our goal is to compute the sum

$$xy = x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0 \quad (1)$$

- Consider another multiplication

$$p = (x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$$

where we observe $x_1y_0 + x_0y_1 = p - x_1y_1 - x_0y_0$

- So, to get the three components in the sum (1), we only need the three multiplications of $n/2$ -sequences:

$$x_1y_1, x_0y_0, \text{ and } p = (x_1 + x_0)(y_1 + y_0)$$

by letting $x_1y_0 + x_0y_1 = p - x_1y_1 - x_0y_0$

- And then we can get xy with only three recursive calls!

Pseudocode (Improved)

Recursive-Multiply(x, y)

1. write $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$
2. compute $x_1 + x_0$ and $y_1 + y_0$
3. $p = \text{Recursive-Multiply}(x_1 + x_0, y_1 + y_0)$
4. $x_1 y_1 = \text{Recursive-Multiply}(x_1, y_1)$
5. $x_0 y_0 = \text{Recursive-Multiply}(x_0, y_0)$
6. **return** $x_1 y_1 \cdot 2^n + (p - x_1 y_1 - x_0 y_0) \cdot 2^{n/2} + x_0 y_0$

Figure 7: Recursive Multiply (Improved)

Time complexity:

- $T(n) = 3T(n/2) + O(n)$ which is $O(n^{1.59})$