

Algorithms-Design and Analysis(Stanford) Notes

zxm

目录

1	Divide and Conquer	2
1.1	Master Method	2
1.2	Counting Inversions	5
1.3	Karatsuba Multiplication	7
1.4	Binary Search	8
1.5	Strassen's Matrix Multiplication	9
1.6	Closest Pair	11
1.7	Fictitious Example	14
2	Sorting & Selection	15
2.1	Merge Sort (D&C)	16

1 Divide and Conquer

1. **DIVIDE** into smaller sub-problems
2. **CONQUER** via recursive calls
3. **COMBINE** solutions of sub-problems into one for the original problem

1.1 Master Method

- Cool feature: a “black-box” method for solving recurrences
- Determine the upper bound of **running time** for most of the D&C algos
- **Assumption:** all sub-problems have equal size
 - unbalanced sub-problems?
 - more than one recurrence?
- **Recurrence format:**
 - base case: $T(n) \leq C$ (a constant), for all sufficiently small n
 - for all larger n , $T(n) \leq aT(\frac{n}{b}) + O(n^d)$
 - * a : # of recurrence calls (e.g., # of sub-problems), $a \geq 1$
 - * b : input size shrinkage factor, $b > 1$, $T(\frac{n}{b})$ is the time required to solve each sub-problem
 - * d : exponent in running time of the combine step, $d \geq 0$
 - * constants a, b, d independent of n
- **Three Cases:**

$$T(n) = \begin{cases} O(n^d \log n), & a = b^d \text{ (case 1)} \\ O(n^d), & a < b^d \text{ (case 2)} \\ O(n^{\log_b a}), & \text{otherwise (case 3)} \end{cases}$$

If $T(n) = aT(\frac{n}{b}) + \Theta(n^d)$, then (with similar proof)

$$T(n) = \begin{cases} \Theta(n^d \log n), & a = b^d \text{ (case 1)} \\ \Theta(n^d), & a < b^d \text{ (case 2)} \\ \Theta(n^{\log_b a}), & \text{otherwise (case 3)} \end{cases}$$

1.1.1 Proof (Recursion Tree Approach)

3 cases \Leftrightarrow 3 types of recursion trees

For simplicity, we assume:

- $T(1) \leq C$ (for some constant C)
- $T(n) \leq aT(\frac{n}{b}) + Cn^d$
- n is a power of b

At each level $j = 0, 1, \dots, \log_b n$, there are

- a^j sub-problems,
- each of size n/b^j
- $(1 + \log_b n)$ levels, level-0: **root**, level- $\log_b n$: **leaves**

Then

$$\text{Total work at level-}j \leq a^j \times C \left(\frac{n}{b^j}\right)^d$$

Thus,

$$\text{Total work} \leq Cn^d \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j \quad (\Delta)$$

\Rightarrow all depends on the relationship between a and b^d

For constant $r > 0$,

$$1 + r + r^2 + \dots + r^k = \frac{1 - r^{k+1}}{1 - r} \leq \begin{cases} \frac{1}{1-r}, & r < 1 \\ \frac{r}{r-1}r^k, & r > 1 \end{cases}$$

- $0 < r < 1$, $LHS \leq \text{Constant}$
- $r > 1$, LHS is dominated by the largest power of r

Case 1. $a = b^d$

$$(\Delta) \leq Cn^d \times \log_b n = O(n^d \log n)$$

Case 2. $a < b^d$

$$(\Delta) \leq O(n^d)$$

Case 3. $a > b^d$

$$(\Delta) \leq Cn^d \left(\frac{a}{b^d}\right)^{\log_b n} = Ca^{\log_b n} = Cn^{\log_b a} = O(n^{\log_b a})$$

$$n^{\log_b a} = a^{\log_b n} \Leftrightarrow \log_b a \log_b n = \log_b n \log_b a$$

1.1.2 Interpretations

Upper bound on the work at level j :

$$Cn^d \times \left(\frac{a}{b^d}\right)^j$$

a : rate of sub-problem poliferation, RSP - force of evil

b^d : rate of work shrinkage (per sub-problem), RWS - force of good

- $RSP < RWS$
 - amount of work is decreasing with the recursion level j
 - most work at root \Rightarrow root dominates \Rightarrow might expect $O(n^d)$
- $RSP > RWS$
 - amount of work is increasing with the recursion level j
 - leaves dominate \Rightarrow might expect $O(\#leaves) = O(a^{\log_b n}) = O(n^{\log_b a})$
- $RSP = RWS$
 - amount of work is the same at each recursion level j (like merge sort)
 - might expect $O(\log n) \times O(n^d) = O(n^d \log n)$ (recursion depth = $O(\log n)$)

1.2 Counting Inversions

- **Problem:** counting # of inversions in an array = # of pairs (i, j) of array indices with $i < j$ and $A[i] > A[j]$
- Number of inversions = Number of intersections of line segments

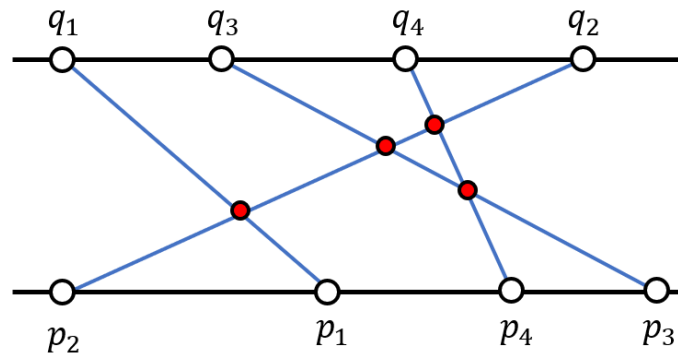


图 1: No. of Inversions = No. of Intersections

- **Application:** measuring similarity between 2 ranked lists => making good recommendations => **collaborative filtering** (CF)
- Largest possible number of inversions that an n -element array can have? C_n^2 (worst case: array in backward order)
- **Brute-force** algorithm: $O(n^2)$
- **D&C** algorithm: pseudocode

```
A = input array [length = n]
D = output [length= n]
B = 1st sorted array [n/2], C = 2nd sorted array [n/2]
```

```
SortAndCount(A)
```

```
if n=1
```

```
    return 0
```

```
else
```

```
    (B,X) = SortAndCount(1st half of A)
```

```
    (C,Y) = SortAndCount(2nd half of A)
```

```
    (D,Z) = MergeAndCountSplitInv(A)
```

```
return X + Y + Z
```

Goal: implement MergeAndCountSplitInv in linear time $O(n)$

The **split inversions** involving an element y of the 2nd array C are precisely the elements left in the 1st array B when y is copied to the output D .

$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = O(n \log n)$$

1.2.1 Python Code

```
def sortAndCount(arr):
    n = len(arr)
    # base case
    if n < 2:
        return arr, 0

    m = n//2
    left, x = sortAndCount(arr[:m])
    right, y = sortAndCount(arr[m:])

    i = j = z = 0
    for k in range(n):
        if j == n - m or (i < m and left[i] < right[j]):
            arr[k] = left[i]
            i += 1
        else:
            arr[k] = right[j]
            j += 1
            z += (m - i)
    return arr, x + y + z
```

1.3 Karatsuba Multiplication

- **Problem:** multiplication of two n -digit numbers x, y (base 10)
- **Application:** cryptography
- Define *primitive operations*: add or multiply 2 single-digit numbers
- Simple method: $\leq 4n^2 = O(n^2)$ operations
- **Recursive method:**

$$x = 10^{n/2}a + b, y = 10^{n/2}c + d, \text{ where } a, b, c, d \text{ are } \frac{n}{2} \text{ - digit numbers}$$

$$\Rightarrow xy = 10^n ac + 10^{n/2}(ad + bc) + bd \quad (*)$$

Algorithm 1 (Naive method)

- recursively compute $ac, ad, bc,$ and bd
- then compute (*)

Running time

$$T(n) \begin{cases} = O(1), & n = 1(\text{base case}) \\ \leq 4T(\frac{n}{2}) + O(n), & n \geq 1(4 \text{ subproblems and linear time bit addition}) \end{cases}$$

By master method,

$$a = 4 > 2^1 = b^d(\text{case 3}) \Rightarrow T(n) = O(n^{\log_b a}) = O(n^2)$$

Algorithm 2 (Gauss method)

- recursively compute $ac, bd, (a + b)(c + d)$
- $ad + bc = (a + b)(c + d) - (ac + bd)$
- then compute (*)

Running time

$$T(n) \begin{cases} = O(1), & n = 1(\text{base case}) \\ \leq 3T(\lceil \frac{n}{2} \rceil) + O(n), & n \geq 1(3 \text{ subproblems and linear time bit addition}) \end{cases}$$

By master method,

$$a = 3 > 2^1 = b^d(\text{case 3}) \Rightarrow T(n) = O(n^{\log_b a}) = O(n^{\log_2 3}) = O(n^{1.59})$$

Better than simple method!

1.3.1 Python Code

```
def karatsuba(x, y):
    nx, ny = len(str(x)), len(str(y))
    n = max(nx, ny)

    # base case
    if n == 1:
        return x*y

    m = n//2
    bm = 10**m
    a, b = x//bm, x%bm
    c, d = y//bm, y%bm
    term1 = karatsuba(a, c)
    term2 = karatsuba(b, d)
    term3 = karatsuba(a+b, c+d)
    result = (bm**2)*term1 + bm*(term3 - term1 - term2) + term2
    return result
```

1.4 Binary Search

- **Problem:** looking for an element in a given sorted array
- **Running time:** $T(n) = T(n/2) + O(1)$. By master method,

$$a = 1 = 2^0 = b^d \text{ (case 1)} \Rightarrow T(n) \leq O(n^d \log n) = O(\log n)$$

1.4.1 Python Code

See [link](#)

1.5 Strassen's Matrix Multiplication

- **Problem:** compute $Z_{n \times n} = X_{n \times n} \cdot Y_{n \times n}$ (Note: input size = $O(n^2)$)
- Naive iterative algorithm: $O(n^3)$ (3 for loops)

$$z_{ij} = \sum_{k=1}^n x_{ik} \cdot y_{kj}$$

- Write

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

where A through H are all $\frac{n}{2} \times \frac{n}{2}$ matrices. Then

$$X \cdot Y = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

- Recursive method #1:
 - **step1.** recursively compute the 8 products
 - **step2.** do additions ($O(n^2)$ time)
 - **Running time:** by master method,

$$a = 8 > 2^2 = b^d \text{ (case 3)} \Rightarrow T(n) \leq O(n^{\log_b a}) = O(n^{\log_2 8}) = O(n^3)$$

- **Strassen's method:**
 - **step1.** recursively compute only 7 (cleverly chosen) products
 - **step2.** do (clever) additions ($O(n^2)$ time)
 - **Running time:** by master method,

$$a = 7 > 2^2 = b^d \text{ (case 3)} \Rightarrow T(n) \leq O(n^{\log_b a}) = O(n^{\log_2 7}) = O(n^{2.81})$$

The 7 products:

$$P_1 = A(F - H)$$

$$P_2 = (A + B)H$$

$$P_3 = (C + D)E$$

$$P_4 = D(G - E)$$

$$P_5 = (A + D)(E + H)$$

$$P_6 = (B - D)(G + H)$$

$$P_7 = (A - C)(E + F)$$

Then

$$\begin{aligned} X \cdot Y &= \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix} \\ &= \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{pmatrix} \end{aligned}$$

```
import numpy as np
def strassen(X, Y):
    n = X.shape[0]
    if n == 1:
        return np.array([X[0, 0]*Y[0, 0]])

    if n%2 == 1: # padding with zeros
        Xpad = np.zeros((n + 1, n + 1))
        Ypad = np.zeros((n + 1, n + 1))
        Xpad[:n, :n], Ypad[:n, :n] = X, Y
        return strassen(Xpad, Ypad)[:n, :n]

    m = n//2
    A, B, C, D = X[:m, :m], X[:m, m:], X[m:, :m], X[m:, m:]
    E, F, G, H = Y[:m, :m], Y[:m, m:], Y[m:, :m], Y[m:, m:]

    P1 = strassen(A, F - H)
    P2 = strassen(A + B, H)
    P3 = strassen(C + D, E)
    P4 = strassen(D, G - E)
    P5 = strassen(A + D, E + H)
    P6 = strassen(B - D, G + H)
    P7 = strassen(A - C, E + F)

    Z = np.zeros((n, n))
    Z11 = P5 + P4 - P2 + P6
    Z12 = P1 + P2
    Z21 = P3 + P4
    Z22 = P1 + P5 - P3 - P7

    Z[:m, :m], Z[:m, m:], Z[m:, :m], Z[m:, m:] = Z11, Z12, Z21, Z22

    return Z
```

1.6 Closest Pair

- **Input:** a set $P = \{p_1, \dots, p_n\}$ of n points in the plane (\mathbb{R}^2)
- Notation: $d(p, q)$ = Euclidean distance
- **Output:** $p^*, q^* = \arg \min_{p \neq q \in P} d(p, q)$
- **Assumption:** (for convenience) all points have distinct x -coordinates, y -coordinates
- **Brute-force search:** $\Theta(n^2)$
- 1-d version of CP:
 - sort points ($O(n \log n)$ time)
 - return CP of adjacent points ($O(n)$ time)
- **Goal:** $O(n \log n)$ time algo for 2-d version
- **D&C:** make copies of points sorted by x -coordinates (P_x) and by y -coordinates (P_y) [$O(n \log n)$ time]

ClosestPair(P_x, P_y)

1. Q = left half of P , R = right half of P . Form Q_x, Q_y, R_x, R_y [$O(n)$ time]
2. $(p_1, q_1) = \text{ClosestPair}(Q_x, Q_y)$
3. $(p_2, q_2) = \text{ClosestPair}(R_x, R_y)$
4. let $\delta = \min\{d(p_1, q_1), d(p_2, q_2)\}$
5. $(p_3, q_3) = \text{ClosestSplitPair}(P_x, P_y, \delta)$. Requirements:
 - (a) $O(n)$ time
 - (b) correct whenever CP of P is a split pair
6. return best of $(p_1, q_1), (p_2, q_2), (p_3, q_3)$

ClosestSplitPair(P_x, P_y, δ) **filtering + linear scan**

1. Let \bar{x} = biggest x -coordinate in the left of P [$O(1)$ time]
2. Let S_y = points of P with x -coordinate in $[\bar{x} - \delta, \bar{x} + \delta]$, sorted by y -coordinate
3. Initialize best = δ , best point = NULL, $m = |S_y|$.

For $i = 1$ to $m-1$

 For $j = 1$ to $\min(7, m-i)$

 Let p, q = i th, $(i+j)$ th points of P

 If $d(p, q) < \text{best}$

 best point = (p, q) , best = $d(p, q)$

[$O(n)$ time]

Claim. Let $p \in Q, q \in R$ be a split pair with $d(p, q) < \delta$. Then

1. $p, q \in S_y$
2. p, q are at most 7 positions apart in S_y

Proof.

(A) Let $p = (x_1, y_1) \in Q, q = (x_2, y_2) \in R$ and $d(p, q) < \delta$. Thus, $|x_1 - x_2| < \delta$.

$$\Rightarrow x_1 \leq \bar{x} \leq x_2 \Rightarrow \begin{cases} \bar{x} - x_1 \leq x_2 - x_1 < \delta & \Rightarrow x_1 > \bar{x} - \delta \\ x_2 - \bar{x} \leq x_2 - x_1 < \delta & \Rightarrow x_2 < \bar{x} + \delta \end{cases}$$

(B) Evenly divide $[\bar{x} - \delta, \bar{x} + \delta] \times [\tilde{y}, \tilde{y} + \delta]$ into $8 \frac{\delta}{2} \times \frac{\delta}{2}$ boxes, where $\tilde{y} = \min\{y_1, y_2\}$.

Then at most 1 point in each of the 8 boxes.

Proof by contradiction.

Let a, b be in the same box. Then $a, b \in Q$ or $a, b \in R$.

$$d(a, b) \leq \frac{\delta}{\sqrt{2}} < \delta. \text{ contradiction to definition of } \delta.$$

Corollary 1. If CP of P is a split pair, then `ClosestSplitPair` can find it.

Corollary 2. `ClosestPair` is correct, and runs in $O(n \log n)$ time.

1.6.1 Python Code

```
import numpy as np
import numpy.linalg as LA
# find closet pair of n 2-dim points
# use n * 2 dim np array P
# each row: 1 point

def CP(P):
    Px = P[P[:, 0].argsort(kind = 'mergesort')]
    Py = P[P[:, 1].argsort(kind = 'mergesort')]
    return closestPair(Px, Py)

def closestPair(Px, Py):
    n = Px.shape[0] # no of points
    if n < 4:
        return bruteForceCP(Px)

    mid = n//2
    Qx, Rx = Px[:mid], Px[mid:]
    midx = Px[mid, 0]
```

```

idx = (Py[:, 0] < midx)
Qy, Ry = Py[idx], Py[~idx]
(p1, q1), r1 = closestPair(Qx, Qy)
(p2, q2), r2 = closestPair(Rx, Ry)

if r1 < r2:
    best = r1
    bestpoint = (p1, q1)
else:
    best = r2
    bestpoint = (p2, q2)

(p3, q3), r3 = closestSplitPair(Px, Py, best, bestpoint)
if r3 < best:
    best = r3
    bestpoint = (p3, q3)
return bestpoint, best

def bruteForceCP(P):
    best = float('Inf')
    bestpoint = None
    n = P.shape[0]
    for i in range(n):
        for j in range(i + 1, n):
            d = LA.norm(P[i] - P[j])
            if d < best:
                best = d
                bestpoint = (P[i], P[j])
    return bestpoint, best

def closestSplitPair(Px, Py, best, bestpoint):
    n = Px.shape[0]
    mid = n//2
    midx = Px[mid, 0]
    idx = np.logical_and(midx - best <= Py[:, 0], Py[:, 0] <= midx + best)
    Sy = Py[idx]

    m = Sy.shape[0] # no of points in Sy
    for i in range(m - 1):
        for j in range(i + 1, min(i + 8, m)):
            p, q = Sy[i], Sy[j]

```

```

        d = LA.norm(p - q)
        if d < best:
            best = d
            bestpoint = (p, q)
    return bestpoint, best

# Test
import time
a = np.random.randint(1e5, size = (1000, 2))
tic = time.time()
b = bruteForceCP(a)
toc = time.time()
print("Brute Force takes time:", toc - tic)
print("ans:", b)
# Brute Force takes time: 4.079403400421143
# ans: ((array([29326, 36462]), array([29246, 36561])), 127.283148923964)

tic = time.time()
c = CP(a)
toc = time.time()
print("D&C takes time:", toc - tic)
print("ans:", c)
# D&C takes time: 0.14911723136901855
# ans: ((array([29326, 36462]), array([29246, 36561])), 127.283148923964)

```

1.7 Fictitious Example

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n^2) \Rightarrow a = 2 < 2^2 = b^d (\text{case 2}) \Rightarrow T(n) = O(n^2)$$

Recursion depth = $O(\log n)$, each level $O(n^2)$.

$T(n) = O(n^2) = o(n^2 \log n)$! Master method gives **a tighter upper bound!**

2 Sorting & Selection

	SORTING	SELECTION
Randomized	Quick Sort	RSelect
Deterministic	Merge Sort	DSelect

Merge Sort versus Quick Sort

- Quick Sort has a smaller constant than Merge Sort

Running Time	<i>Average Case</i>	<i>Worst Case</i>
<i>Merge Sort</i>	$O(n \log n)$	$O(n \log n)$
<i>Quick Sort</i>	$O(n \log n)$	$O(n^2)$

2.1 Merge Sort (D&C)

- **Input:** array of numbers (assume distinct, **exercise:** duplicate case), unsorted
- **Output:** same numbers, sorted in increasing order
- **Pseudocode for Merge:**

```

C = output [length=n]
  A = 1st sorted array [n/2], B = 2nd sorted array [n/2]
  i = 1, j = 1

for k = 1 to n
  if A(i)<B(j)
    C(k)=A(i)
    i++
  else [B(j)<A(i)]
    C(k)=B(j)
    j++
end

(ignore end cases)

```

2.1.1 Running Time

- **Key question:** running time of **Merge Sort** on array of n numbers?
- **Upshot:** running time of **Merge** on an array of m numbers $\leq 5m + 2 \leq 6m (m > 1)$

$i = 1, j = 1$ [2 operations]

each **for** loop: comparison, assignment, increment of i or j , increment of k , comparison of k to the upper bound n

- **Claim:** Merge Sort requires $\leq 6n \log_2 n + 6n$ operations to sort n numbers.

2.1.1.1 Proof by Recursion Tree Method

- Assume $n =$ power of 2
- At each level $j = 0, 1, 2, \dots, \log_2 n$, there are 2^j sub-problems, each of size $n/2^j$.
- Total # of operations at level j (**Merge**):

$$\leq 2^j \times 6 \left(\frac{n}{2^j} \right) = 6n \text{ (independent of } j \text{)}$$

- Total # of operations $\leq 6n(\log_2 n + 1)$

2.1.1.2 Proof by Master Method

$$a = 2 = 2^1 = b^d \text{ (case 1)} \Rightarrow T(n) \leq O(n^d \log n) = O(n \log n)$$

2.1.2 Python Code

```
def mergeSort(arr):
    n = len(arr)
    # base case
    if n < 2:
        return arr[:]

    m = n//2
    left, right = mergeSort(arr[:m]), mergeSort(arr[m:])
    left.append(float('Inf'))
    right.append(float('Inf'))
    i = j = 0
    result = []
    for k in range(n):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    return result
```

- Space complexity: $O(n)$