AI2615 算法设计与分析 2020-2021 春季学期

Lecture 3 – Divide and Conquer

2021 年 3 月 12 日

Lecturer: 张驰豪 *Scribe:* 陶表帅

Divide and conquer is a common algorithm design technique that recursively divides a problem instance into multiple sub-instances of the same problem, until the sub-instances are small enough to be solved directly. The idea behinds divide and conquer is similar to mathematical induction. In fact, mathematical induction is normally applied to prove the correctness of a divide-and-conquer-based algorithm. Finding clever ways to divide the problem instance and to merge the solutions of multiple sub-instances is usually the key to make a divide-and-conquer-based algorithm run faster. In this lecture, we will consider some classical problems that can be typically solved by divide and conquer.

1 Sorting an Array

Problem 1. Given an array $a[1,\ldots,n]$ of n integers, sort the array in the ascending order.

We introduce a divide and conquer algorithm, the *merge sort*. Merge sort divides the array $a[1,\ldots,n]$ into two arrays, $a\left[1,\ldots,\lfloor\frac{n}{2}\rfloor\right]$ and $a\left[\lfloor\frac{n}{2}\rfloor+1,\ldots,n\right]$, sorts the two arrays recursively, and then merges the two sorted arrays. The algorithm is described in Algorithm 1.

Algorithm 1 The merge sort

MergeSort($a[1,\ldots,n]$):

1: **if** $n=1$, **return** a
2: $b \leftarrow$MergeSort$\left(a\left[1,\ldots,\lfloor\frac{n}{2}\rfloor\right]\right)$
3: $c \leftarrow$MergeSort$\left(a\left[\lfloor\frac{n}{2}\rfloor+1,\ldots,n\right]\right)$
4: **return** Merge(b,c)

It remains to define the function Merge that merges two sorted arrays b and c. This can be done easily, as described in Algorithm 2.

1.1 Time Complexity of Merge Sort

We first analyze the time complexity for computing the function Merge$(b[1,\ldots,m],c[1,\ldots,n])$. Notice that exactly one element is added into a at each while-loop iteration, and each element in b and c is added exactly once. It is easy to see that the time complexity here is $O(m+n)$.

Algorithm 2 The MERGE function

MERGE($b[1,\ldots,m], c[1,\ldots,n]$): // b and c are sorted in ascending order

initialize array a with length 0
initialize $i \leftarrow 1$ and $j \leftarrow 1$
while $i \le m$ and $j \le n$:
    if $b[i] \le c[j]$:
        a.push_back($b[i]$)
        $i \leftarrow i + 1$
    else
        a.push_back($c[j]$)
        $j \leftarrow j + 1$
    endif
endwhile
if $i > m$, append $c[j,\ldots,n]$ to the end of a
if $j > n$, append $b[i,\ldots,m]$ to the end of a
return a

To analyze the time complexity of MERGESORT, let $T(n)$ be the complexity of MERGESORT when the length of the input array is n. To simplify the analysis, we assume n is an integer power of 2 (i.e., $n = 2^k$ for some $k \in \mathbb{Z}^+$) without loss of generality[1]. The time complexity for sorting the first half of the array $a\left[1,\ldots,\frac{n}{2}\right]$ is then $T(\frac{n}{2})$, and the time complexity for sorting the second half of the array $a\left[\frac{n}{2}+1,\ldots,n\right]$ is also $T(\frac{n}{2})$. Finally, it takes $O(\frac{n}{2}+\frac{n}{2}) = O(n)$ times to merge the two arrays as we have analyzed before. Therefore, we obtain the following recurrence relation on T:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n), \qquad \text{where } T(1) = 1.$$

To solve this recurrence relation, let $c > 0$ be a constant such that $T(n) \le 2T(\frac{n}{2}) + cn$. By iteratively applying

[1] If n is not an integer power of 2, we can find the largest integer s in the array, add more integers that are even larger than s into the array to make the length of the array an integer power of 2, and then remove those newly added integers after sorting the array. We leave it as an exercise to show that the time complexity does not change asymptotically by these operations.

the recurrence relation, we have

$$\begin{aligned} T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\ &\le 2\left(2T\left(\frac{n}{4}\right) + c\cdot\frac{n}{2}\right) + cn = 4T\left(\frac{n}{4}\right) + 2cn \\ &\le 4\left(2T\left(\frac{n}{8}\right) + c\cdot\frac{n}{4}\right) + 2cn = 8T\left(\frac{n}{8}\right) + 3cn \\ &\quad\vdots \\ &\le n\cdot T(1) + (\log_2 n)\cdot cn = O(n\log n). \end{aligned}$$

Putting together, we have completed the analysis of the time complexity for the merge sort.

Theorem 2. *The time complexity for sorting an array of length n by the merge sort is $O(n\log n)$.*

1.2 Is $\Theta(n\log n)$ Optimal?

We have seen that the merge sort can sort an array in time $O(n\log n)$. Can we do better? In other words, does there exist an algorithm that runs in time $o(n\log n)$?

To answer this question, we first need to carefully define what *computational model* we are using. *Computational complexity-theoretic Church–Turing thesis* posits that the class of problems that are solvable in polynomial time in other "reasonable" computational models is also polynomial time solvable in a Turing machine. However, when it comes to use a specific polynomial to upper-bound the time complexity of a given algorithm, the precise polynomial upper bounds can be different for different computational models. When analyzing the time complexity for the merge sort, we have implicitly assumed that comparing the values of two integers can be done in $O(1)$ time, which is true in the RAM (Random Access Machine) model, but unrealistic in a Turing Machine. Other than the Turing Machine model, a commonly used model that arguably closer to modern computers is called *word RAM*, where arithmetic operations such as additions and multiplications can be done in $O(1)$ time.

Under the word RAM model, the current fastest randomized sorting algorithm runs in time $O(n\sqrt{\log\log n})$ due to Han and Thorup [2002], and the current fastest deterministic sorting algorithm runs in time $O(n\log\log n)$ due to Han [2002].

However, if we are in a computational model where the only available binary operation between two elements is comparison (for example, the elements can be non-numeric, but there is a total order defined), then $\Theta(n\log n)$ is indeed optimal.

Theorem 3. *Suppose the only binary operation allowed is comparison. Any sorting algorithm requires making $\Omega(n\log n)$ comparisons.*

Proof. Let $K(n)$ be the maximum number of comparisons an arbitrary algorithm makes for an array of length n. Since a comparison can only have 3 outcomes: greater than, equal to, or smaller than, the number of possible outputs by the algorithm is at most $3^{K(n)}$.

On the other hand, an array of length n can have $n!$ different orders. Given the ordered set of indices $(1,\ldots,n)$, each output of the algorithm corresponds to a permutation of $(1,\ldots,n)$. If the output is characterized by the permutation, the algorithm must be able to produce $n!$ different outputs.
Putting together, we have $3^{K(n)} \geq n!$, which implies $K(n) = \Omega(\log n!) = \Omega(n \log n)$. We have applied the inequality $(\frac{n}{2})^{n/2} \leq n! \leq n^n$.[2] □

2 Counting the Number of Inversions

For an array $a[1,\ldots,n]$ and a pair of indices (i,j), we say that (i,j) is an *inversion* if $i < j$ and $a[i] > a[j]$. An inversion is a pair that disobeys the ascending order.

Problem 4. Given an array $a[1,\ldots,n]$ that is a permutation of $(1,\ldots,n)$, count the number of inversions in a, $\#\{(i,j) \mid (i<j) \wedge (a[i] > a[j])\}$.

To apply the divide and conquer technique, we split the array into two sub-arrays: $a\left[1,\ldots,\lfloor\frac{n}{2}\rfloor\right]$ and $a\left[\lfloor\frac{n}{2}\rfloor+1,\ldots,n\right]$ as before. Suppose x_a is the number of inversions within $a\left[1,\ldots,\lfloor\frac{n}{2}\rfloor\right]$ and x_b is the number of inversions within $a\left[\lfloor\frac{n}{2}\rfloor+1,\ldots,n\right]$. The total number of inversions is $x_a + x_b$ plus the number of those inversions (i,j) where index i comes from the sub-array $a\left[1,\ldots,\lfloor\frac{n}{2}\rfloor\right]$ and index j comes from the sub-array $a\left[\lfloor\frac{n}{2}\rfloor+1,\ldots,n\right]$. The values of x_a and x_b are computed recursively, so it remains to compute the number of inversions between the two sub-arrays.
Notice that the indices of elements in $a\left[\lfloor\frac{n}{2}\rfloor+1,\ldots,n\right]$ are higher than the indices of elements in $a\left[1,\ldots,\lfloor\frac{n}{2}\rfloor\right]$. We only need to find all pairs of integers, one from each sub-array, such that the integer from the sub-array $a\left[1,\ldots,\lfloor\frac{n}{2}\rfloor\right]$ is greater than the integer from the sub-array $a\left[\lfloor\frac{n}{2}\rfloor+1,\ldots,n\right]$. This can be easily done if the two sub-arrays are sorted, but how can we guarantee the two sub-arrays are sorted? The trick here is that sorting and counting the number of inversions can be done *simultaneously*. The algorithm is presented in Algorithm 3.

2.1 Time Complexity for Counting the Number of Inversions

As before, let $T(n)$ be the time complexity for Algorithm 3 when the array has length n, and we aim to build a recurrence relation for $T(n)$. We have seen that the computation of function Merge at Line 15 requires time $O(n)$. By a similar analysis, the computations from Line 4 to Line 14 requires time $O(n)$. Putting together, we have

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n), \qquad \text{where } T(1) = 1.$$

We know that this gives us $T(n) = O(n \log n)$ by the computation in the previous section.

[2] The part $n! \leq n^n$ can be easily seen by term-wise comparisons of the n terms. The part $(\frac{n}{2})^{n/2} \leq n!$ can be seen by $(\frac{n}{2})^{n/2} \leq (\frac{n}{2}+1)(\frac{n}{2}+2)\cdots n \leq n!$.

Algorithm 3 Counting the number of inversions

CountInversions($a[1,\dots,n]$)

1: **if** $n = 1$, **return** $(0, a)$ // We make sure CountInversions output both the number of inversions *and* the sorted array.
2: $(x_b, b) \leftarrow$countInversions$(a[1,\dots,\lfloor\frac{n}{2}\rfloor])$
3: $(x_c, c) \leftarrow$countInversions$(a[\lfloor\frac{n}{2}\rfloor+1,\dots,n])$
4: $x \leftarrow x_b + x_c$ // below we compute the number of inversions between b and c
5: initialize $i \leftarrow 1$ and $j \leftarrow 1$
6: **while** $i \le \lfloor\frac{n}{2}\rfloor$ and $j \le n - \lfloor\frac{n}{2}\rfloor$:
7: **if** $b[i] \le c[j]$:
8: $i \leftarrow i + 1$
9: $x \leftarrow x + j - 1$
10: **else**
11: $j \leftarrow j + 1$
12: **endif**
13: **endwhile**
14: **if** $j > n - \lfloor\frac{n}{2}\rfloor$, update $x \leftarrow x + (n - \lfloor\frac{n}{2}\rfloor)(\lfloor\frac{n}{2}\rfloor - i + 1)$
15: $a \leftarrow$Merge(b, c) // Merge is defined in Algorithm 2. This step ensures sub-arrays are sorted.
16: **return** (x, a).

Theorem 5. *The time complexity of Algorithm 3 is* $O(n \log n)$.

We ask the same question as before: can we do better? Under the word RAM model, the current state-of-the-art is an algorithm runs in time $O(n\sqrt{\log n})$ by Chan and Pătraşcu [2010].

3 Master Theorem

When we apply the divide and conquer technique, we usually arrive at a recurrence relation

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

where a, b and d are constants independent of n. *Master theorem* below is a tool to solve the recurrence relations in this format.

Theorem 6 (Master Theorem). *Given the recurrence relation* $T(n) = aT\left(\frac{n}{b}\right) + O\left(n^d\right)$ *where* $a, b, d > 0$ *are constants independent of* n *and the initial condition satisfies* $T(1) = O(1)$, *we have*

$$T(n) = \begin{cases} T(n) = O(n^d) & \text{if } d > \log_b a \\ T(n) = O(n^d \log n) & \text{if } d = \log_b a \\ T(n) = O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}.$$

Proof. Let c be a constant satisfying $T(n) \le aT\left(\frac{n}{b}\right) + cn^d$. We have

$$\begin{aligned}
T(n) &\le aT\left(\frac{n}{b}\right) + cn^d \\
&\le a\left(aT\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^d\right) + cn^d = a^2T\left(\frac{n}{b^2}\right) + cn^d \cdot \left(1 + \frac{a}{b^d}\right) \\
&\le a^2\left(aT\left(\frac{n}{b^3}\right) + c\left(\frac{n}{b^2}\right)^d\right) + cn^d \cdot \left(1 + \frac{a}{b^d}\right) = a^3T\left(\frac{n}{b^3}\right) + cn^d \cdot \left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2\right) \\
&\quad \vdots \\
&\le a^{\log_b n}T(1) + cn^d \cdot \sum_{i=0}^{\log_b n - 1}\left(\frac{a}{b^d}\right) = O\left(n^{\log_b a}\right) + cn^d \cdot \sum_{i=0}^{\log_b n - 1}\left(\frac{a}{b^d}\right).
\end{aligned}$$

If $d > \log_b a$, we have $\frac{a}{b^d} < 1$ and $\sum_{i=0}^{\log_b n - 1}\left(\frac{a}{b^d}\right) = O(1)$. Therefore, $T(n) = O(n^d)$.
If $d = \log_b a$, we have $\frac{a}{b^d} = 1$ and $\sum_{i=0}^{\log_b n - 1}\left(\frac{a}{b^d}\right) = \log_b n$. Therefore, $T(n) \le n^d T(1) + cn^d \cdot \log_b n = O(n^d \log n)$.
If $d < \log_b a$, we have $\frac{a}{b^d} > 1$ and $\sum_{i=0}^{\log_b n - 1}\left(\frac{a}{b^d}\right) = O\left(\left(\frac{a}{b^d}\right)^{\log_b n}\right) = O\left(\frac{n^{\log_b a}}{n^d}\right)$. Therefore, $T(n) = O(n^{\log_b a})$. □

4 Integer Multiplications in Turing Machine Model

In this section, we consider the multiplication of two n-bit integers a and b in Turing Machine model.

Problem 7. Given two n-bit integers a and b, compute the product ab.

For simplicity, we assume n is an integer power of 2 (notice that we can always append zeros to the most significant bits without change the values of a and b until n is an integer power of 2).
To use the divide and conquer technique, we split each of a and b to two parts: the "left" part with $n/2$ most significant bits, and the "right" part with $n/2$ least significant bits. We write

$$a = a_L \times 2^{\frac{n}{2}} + a_R \qquad \text{and} \qquad b = b_L \times 2^{\frac{n}{2}} + b_R,$$

and we have

$$ab = a_L b_L \times 2^n + (a_L b_R + a_R b_L) \times 2^{\frac{n}{2}} + a_R b_R.$$

We aim to design a divide-and-conquer-based algorithm based on this equation. Notice that additions can be done in linear time in a Turing Machine. Multiplications by 2^n and $2^{\frac{n}{2}}$ can also be done in linear time, as multiplications by an integer power of 2 only require shifting all the bits of the integers to the left. The four multiplications $a_L b_L, a_L b_R, a_R b_L, a_R b_R$ can be done recursively.
Based on these, we have a divide-and-conquer-based algorithm with the time complexity characterized by $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$, which, by master theorem, gives us $T(n) = O(n^2)$. However, this is no better than the naïve primary school bit-wise integer multiplication method.

However, a trick from Carl Friedrich Gauss reduces the number of multiplications in each recursive step from 4 to 3. We only compute three multiplications $a_L b_L, a_L b_R$ and $(a_L + a_R)(b_L + b_R)$, and the coefficient of $2^{\frac{n}{2}}$ can be computed by only additions/subtractions:

$$a_L b_R + a_R b_L = (a_L + a_R)(b_L + b_R) - a_L b_L - a_L b_R.$$

The idea behinds this is that multiplications are much more computational expensive than additions, so it is worthy to reduce the number of multiplications at the cost of increasing the number of additions/subtractions.

The divide and conquer algorithm by applying Gauss's trick has time complexity characterized by the recurrence relation $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$, which yields $T(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$ by master theorem. The algorithm is formally described in Algorithm 4.

Algorithm 4 Algorithm for integer multiplications

MULTIPLY(a, b):

1: write $a = a_L \times 2^{\frac{n}{2}} + a_R$ and $b = b_L \times 2^{\frac{n}{2}} + b_R$
2: $x \leftarrow$MULTIPLY(a_L, b_L)
3: $y \leftarrow$MULTIPLY(a_R, b_R)
4: $z \leftarrow$MULTIPLY$(a_L + a_R, b_L + b_R)$
5: **return** $x2^n + (z - x - y)2^{\frac{n}{2}} + y$

Theorem 8. *The time complexity for Algorithm 4 is* $O(n^{\log_2 3}) \approx O(n^{1.59})$.

There exist algorithms that run faster. We will learn algorithm Fast Fourier Transform (FFT) in the future, which runs in time $O^*(n \log n)$, where the $*$ in the superscript indicates that those $o(\log n)$ multiplicative factors are omitted from $n \log n$. Harvey and Van Der Hoeven [2021] improved this to $O(n \log n)$.

5 Matrix Multiplications

Problem 9. Given two matrices $X, Y \in \mathbb{Z}^{n \times n}$, compute their product XY.

For simplicity, we again assume n is an integer power of 2. The entry (i, j) of XY is defined by

$$(XY)_{i,j} = \sum_{k=1}^{n} X_{ik} \cdot Y_{kj}.$$

The naïve algorithm to compute the product XY takes $O(n^3)$ time, as we need to compute n^2 entries and each entry takes $O(n)$ time to compute. Can we do it faster by the divide and conquer technique?

Naturally, we can split each of X and Y into four sub-matrices of dimension $\frac{n}{2} \times \frac{n}{2}$ as follows:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad \text{and} \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix},$$

and we have

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

The addition of two matrices takes $O(n^2)$ time, and the eight multiplications above are computed recursively. This gives us a divide-and-conquer-based algorithm with time complexity characterized by $T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$. By master theorem, this gives us $T(n) = O(n^3)$, which is no faster than the naïve algorithm.

Volker Strassen successfully reduced the number of multiplications in each recursive step from 8 to 7. Although the idea of reducing the number of multiplications at the cost of more less-expensive additions/subtractions is similar to Gauss's trick in the previous section, Volker Strassen's solution is much more tricky:

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix},$$

where

$$\begin{aligned} P_1 &\triangleq A(F - H) & P_2 &\triangleq (A + B)H & P_3 &\triangleq (C + D)E \\ P_4 &\triangleq D(G - E) & P_5 &\triangleq (A + D)(E + H) & P_6 &\triangleq (B - D)(G + H) \\ & & P_7 &\triangleq (A - C)(E + F). \end{aligned}$$

The running time of Strassen's algorithm is

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2),$$

which, by master theorem, yields $T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$.

Theorem 10. *The time complexity for Strassen's algorithm is $O(n^{\log_2 7}) \approx O(n^{2.81})$.*

Can we do better? This problem is so significant that theoretical computer scientists reserve a letter ω to denote the exponent of n in the running time of the fastest matrix multiplication algorithm. As shown just now, Strassen's algorithm implies $\omega \leq \log_2 7$. On the other hand, we know $\omega \geq 2$ as it takes time $\Theta(n^2)$ to just read the two input matrices. The current best algorithm by Alman and Williams [2021] indicates $\omega < 2.37286$. It is conjectured by the theoretical computer scientists that $\omega < 2 + \varepsilon$ for any $\varepsilon > 0$.

6 Finding the k-th Smallest Element in an Array

Problem 11. Given an array $a[1, \ldots, n]$, compute the k-th smallest element in a.

We can surely sort the array a and then output $a[k]$. The time complexity for this is $O(n \log n)$ as shown in Sect. 1. However, can we do better? It turns out that there exists an algorithm runs in time $O(n)$. Notice that this is already the best we can do, as it takes $\Theta(n)$ time to read the input. In this section, we present a randomized algorithm based on the divide and conquer technique with expected running time $O(n)$.

We assume for simplicity that all the elements in a are distinct. The algorithm is presented in Algorithm 5. We leave it as an exercise to generalize Algorithm 5 to the scenario where elements in a need not to be distinct.

Algorithm 5 Finding the k-th smallest element

FIND(a, k):

```
1: choose a pivot element x in a uniformly at random
2: compute the rank ℓ of x, i.e., the value ℓ such that exactly ℓ − 1 elements in a are less than x
3: if ℓ = k, return x
4: if ℓ > k:
5:     compute the array a_{<x} consisting of all elements in a that are less than x
6:     return FIND(a_{<x}, k)
7: else
8:     compute the array a_{>x} consisting of all elements in a that are greater than x
9:     return FIND(a_{>x}, k − ℓ)
10: endif
```

It is easy to check the correctness of the algorithm.

For the time complexity, it is easy to check that the operation in each of Line 2, 5 and 8 takes $O(n)$ time. Noticing that the length of $a_{<x}$ is $\ell - 1$ and the length of $a_{>x}$ is $n - \ell$, the time complexity of Algorithm 5 is characterized by

$$T(n) \le T(\max\{n - \ell, \ell - 1\}) + O(n),$$

where ℓ is a number chosen uniformly at random from $\{1, \dots, n\}$.

The worst-case running time for Algorithm 5 is $O(n^2)$, which happens, for example, when $k = n$ and the pivot element chosen happens to be the smallest element in a in each recursive step. What is the average-case running time, or the expected running time, of Algorithm 5?

Let X_i be the random variable describing the length of the array in the problem instance at i-th level of recursive step. In particular, we have $X_1 = n$ and $X_2 = \max(\max\{n - \ell, \ell - 1\})$ for a randomly chosen $\ell \in_R \{1, \dots, n\}$. Notice that the maximum level of recursion is n. If the algorithm terminates at the t-th level, we let $X_i = 0$ for all $i > t$. Let c be the constant such that $T(n) \le T(\max\{n - \ell, \ell - 1\}) + cn$. We then have

$$\begin{aligned}
\mathbb{E}[T(n)] &\le \mathbb{E}[T(X_2) + cX_1] = \mathbb{E}[T(X_2)] + c \cdot \mathbb{E}[X_1] \\
&\le \mathbb{E}[T(X_3) + cX_2] + c\mathbb{E}[X_1] = \mathbb{E}[T(X_3)] + c\,(\mathbb{E}[X_1] + \mathbb{E}[X_2]) \\
&\quad \vdots \\
&\le \mathbb{E}[T(X_n)] + c\,(\mathbb{E}[X_1] + \cdots + \mathbb{E}[X_{n-1}])\,,
\end{aligned}$$

so

$$\mathbb{E}[T(n)] = O\left(\sum_{i=1}^{n} \mathbb{E}[X_i]\right).$$

The expectation of X_2 can be computed by the law of total expectation:

$$\begin{aligned}
\mathbb{E}[X_2 \mid X_1 = n] &= \mathbb{E}[\max(\{n-\ell, \ell-1\})] \\
&= \mathbb{E}[\max(\{n-\ell, \ell-1\}) \mid n-\ell \geq \ell-1]\,\mathbf{Pr}[n-\ell \geq \ell-1] \\
&\quad + \mathbb{E}[\max(\{n-\ell, \ell-1\}) \mid n-\ell < \ell-1]\,\mathbf{Pr}[n-\ell < \ell-1] \\
&= \mathbb{E}\left[n-\ell \mid \ell \leq \frac{n+1}{2}\right]\mathbf{Pr}[n-\ell \geq \ell-1] + \mathbb{E}\left[\ell-1 \mid \ell > \frac{n+1}{2}\right]\mathbf{Pr}[n-\ell < \ell-1] \\
&\leq \frac{3}{4}n\,(\mathbf{Pr}[n-\ell \geq \ell-1] + \mathbf{Pr}[n-\ell < \ell-1]) \qquad (\dagger) \\
&= \frac{3}{4}n.
\end{aligned}$$

For step (†), notice that a number chosen uniformly at random from the first half of $\{1, \ldots, n\}$ has expectation $n/4$ and the number chosen uniformly at random from the second half has expectation $3n/4$.
In general, we have $\mathbb{E}[X_{t+1} \mid X_t] \leq \frac{3}{4}X_t$. Taking expectation on both side yields

$$\mathbb{E}[X_{t+1}] \leq \frac{3}{4}\mathbb{E}[X_t] \leq \cdots \leq \left(\frac{3}{4}\right)^t n.$$

Putting together,

$$\mathbb{E}[T(n)] = O\left(\sum_{i=1}^{n} \mathbb{E}[X_i]\right) = O\left(n \cdot \sum_{i=0}^{n-1} \left(\frac{3}{4}\right)^i\right) = O(n).$$

Theorem 12. *The average-case running time for Algorithm 5 is $O(n)$.*

7 Finding Closest Pair of Points

Problem 13. Given a set of n points $S = \{(x_i, y_i) \mid i = 1, \ldots, n\}$ in $\mathbb{R}^2$, find two points (x_i, y_i) and (x_j, y_j) with the smallest Euclidean distance $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

To begin applying the divide and conquer technique, we find a vertical pivotal line $x = t$ such that half of the points are on the left-hand side of $x = t$ and half of the points are on the right. This can be done by sorting the n points by their x-coordinates and taking $t = \frac{1}{2}\left(x_{\lfloor \frac{n}{2} \rfloor} + x_{\lfloor \frac{n}{2} \rfloor + 1}\right)$. Notice that we only need to sort the points once at the beginning. We do not need to do the sorting at every recursive step.
Let S_L be the set of all points on the left and S_R be the set of all points on the right. We can find the distance between two closest points in each of S_L and S_R recursively. Let h_L and h_R be the corresponding distances for the two closest points in S_L and the two closest points in S_R respectively. The minimum distance is then at most $\min\{h_L, h_R\}$. However, we have not checked for those pairs of points "across the boundary $x = t$". That is, there may be a point in S_L and a point in S_R whose distance is smaller than $\min\{h_L, h_R\}$.

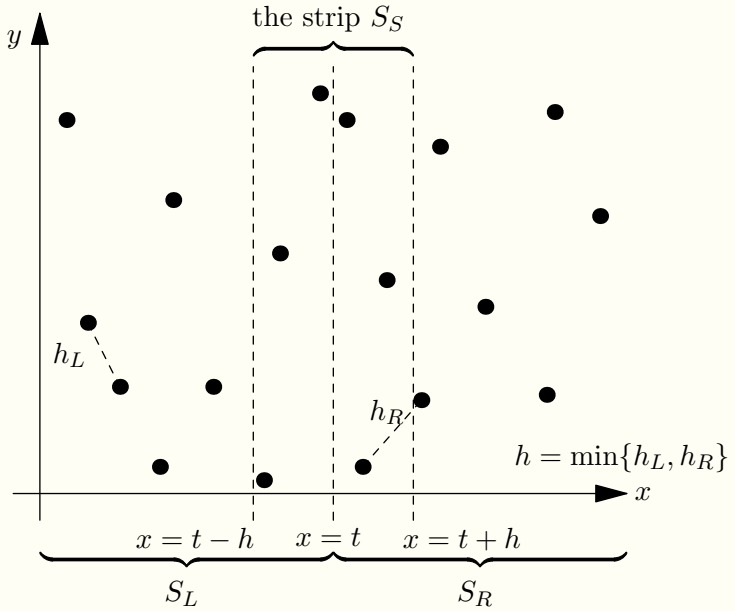

Figure 1: Illustration for S_L, S_R and the strip S_S.

Let $h = \min\{h_L, h_R\}$. We only need to worry about those points that are within the vertical "strip" with left boundary $x = t - h$ and right boundary $x = t + h$. In other words, we only need to consider the set of points $\{(x_i, y_i) \mid x_i \in (t - h, t + h)\}$. Let $S_S = \{(x_i, y_i) \mid x_i \in (t - h, t + h)\}$ be the set of all points in this strip. Figure 1 illustrates S_L, S_R and the strip S_S. For any point A that are outside this strip, the distance from this point to the line $x = t$ is at least h, so the distance from A to any point on the other side of $x = t$ is at least h. We know that A cannot be in a pair of points across the line $x = t$ that has distance less than h. Let h_S be the distance of the two closest points within the strip. Then the distance of the two closest points in S is $\min\{h_L, h_R, h_S\}$. It remains to calculate h_S.

A naïve way to calculate h_S is to compute the distance between every pair of points within the strip. This takes time $O(n^2)$, which is unacceptable. Instead, we will do the followings.

Firstly, we sort the elements in S_S by the y-coordinates. In each iteration, we consider a point $(x_i, y_i) \in S_S$, and we only need to consider those points in S_S whose y-coordinates are at least y_i (for each $(x_j, y_j) \in S_S$ with $y_j < y_i$, the pair of points $(x_i, y_i), (x_j, y_j)$ has already been considered at (x_j, y_j)-th iteration). In addition, we only need to consider the points whose y-coordinates belong to the interval $[y_i, y_i + h]$, as the distance between (x_i, y_i) and a point with y-coordinate greater than $y_i + h$ is more than h. Figure 2 shows that there can be at most 8 points in S_S whose y-coordinates belong to the interval $[y_i, y_i + h]$. Therefore, we only need to compute the distances between (x_i, y_i) and at most 7 other points. Thus, we need to compute the distance between at most $7|S_S| = O(n)$ pairs of points in the strip.

Now, we put everything together and obtain Algorithm 6.

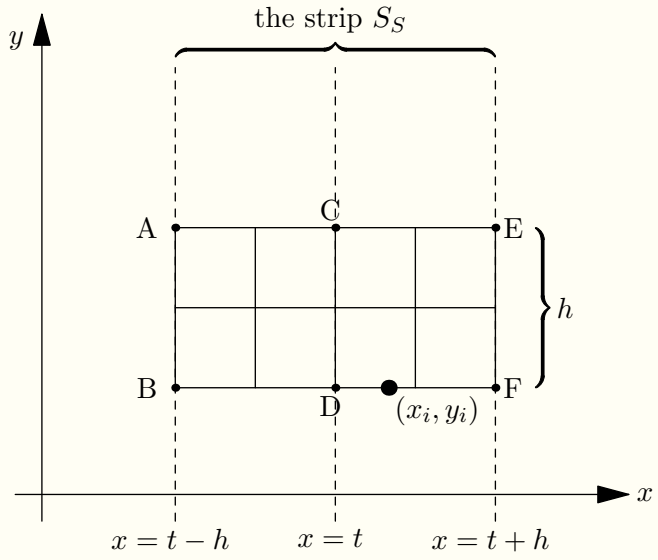

Figure 2: When we are at the (x_i, y_i)-th iteration, we only need to consider those points in the rectangle ABFE. Notice that there can be at most four points in the square ABDC, because there can be at most one point in each of the four small squares in ABDC (notice that any two points in a small square can have distance at most $\frac{\sqrt{2}}{2}h < h$, and the distance between any two points in S_L is at least $h_L \geq h$). Similarly, there can be at most four points in the square CDFE. Thus, there can be at most 8 points in the rectangle ABFE. Excluding (x_i, y_i), there can be at most 7 other points.

Algorithm 6 Finding the distance between two closest points

FindClosest(S): // S is sorted by the ascending order of the x-coordinates

find t such that half of the points in S is on the left hand side of $x = t$
compute $S_L = \{(x_i, x_j) \mid x_i \leq t\}, S_R = \{(x_i, x_j) \mid x_i > t\}$
$h_L \leftarrow$FindClosest(S_L)
$h_R \leftarrow$FindClosest(S_R)
$h \leftarrow \min\{h_L, h_R\}$
compute $S_S = \{(x_i, x_j) \mid x_i \in (t - h, t + h)\}$
sort elements in S_S by the ascending order of the y-coordinates
for each $i = 1, \ldots, |S_S|$:
    for each $j = i + 1, i + 2, \ldots, \min\{i + 7, |S_S|\}$:
        update $h \leftarrow \min\left\{h, \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}\right\}$
    endfor
endfor
return h.

7.1 Time Complexity for Finding the Closest Pair of Points

To analyze the time complexity of Algorithm 6, the time required to execute the for-loop at Line 8 is $O(n)$. The time required to execute the sorting at Line 7 is $O(n \log n)$. However, this can be optimized to $O(n)$ and we leave this optimization as an exercise (see Exercise 15). Therefore, the time complexity is characterized by

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n),$$

which yields $T(n) = O(n \log n)$ by master theorem.

Theorem 14. *The time complexity for Algorithm 6 is $O(n \log n)$.*

Exercise 15. Optimize the execution of the sorting at Line 7 to make Algorithm 6 runs in time $O(n \log n)$.

Can we do better here? A randomized algorithm runs in time $O(n)$ was given by Khuller and Matias [1995].

参考文献

Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. SIAM, 2021. 8

Timothy M Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 161–173. SIAM, 2010. 5

Y Han and M Thorup. Sorting integers in $O(n\sqrt{\log\log n})$ expected time and linear space. In *IEEE Symposium on Foundations of Computer Science (FOCS'02)*, 2002. 3

Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. In *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, pages 602–608, 2002. 3

David Harvey and Joris Van Der Hoeven. Integer multiplication in time $O(n \log n)$. *Annals of Mathematics*, 193(2):563–617, 2021. 7

Samir Khuller and Yossi Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118(1):34–37, 1995. 13