Sufficient Algorithms

Charlotte Xia

November 28, 2023

Contents

1 Fibnacci Problem

1.1 Problem Description

The well known Fibnacci Problem is defined as follows:

Definition 1.1.

$$F_n = \begin{cases} F_{n-1}+F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases} \tag{1}$$

1.2 Solution

1.2.1 Naive Case and Time Complexity

Intuitively, we count the next number by adding the previous two numbers.

Algorithm 1: fib1(n)

```
if n=0: return 0
if n=1: return 1
return fib1(n-1)+fib1(n-2)
```

However, it has an exponential time complexity.

Lemma 1.1. $T_n > F_n$,where T_n stands for the steps needed to calculate F_n.

Proof. Firstly, we have:

$$T(n) = T(n-1) + T(n-2) + 3 \textbf{ for } n > 1$$

where 3 extra steps are needed for 2 'if' judgements and the addition operation.
In the Base Step, for $n = 0$, we have $T_0 = 1 > 0 = F_0$, and for $n = 1$, we have $T_1 = 2 > 1 = F_1$.
In the Inductive Step, suppose $T_i > F_i$ for $i = 1, 2, \dots, n$.

$$\begin{aligned} T_{n+1} &= T_n + T_{n-1} + 3 \\ &> F_n + F_{n-1} + 3 \quad \text{(induction hypothesis)} \\ &= F_{n+1} + 3 \\ &> F_{n+1}. \end{aligned}$$

Therefore, by mathematical induction, we have proved that $T_n > F_n \ \forall n \in N$. $\square$

Furthermore, according to, $F_n = 2^{0.694n} = 1.6^n$, which gives out an exponential time complexity.

1.2.2 Dynamic Programming and Big Number Addition

We notice that every time we calculate F_n, we have to recalculate F_{n-1} and F_{n-2}, which is a waste of time. By creating an array to store the past values, we've come with an polynomial solution ,where approximately, every T_n is obtained by the one step addition of T_{n-1} and T_{n-2}.

However, when n is large enough, the number of digits of F_n will exceed the maximum number of digits that a computer can store(eg. 64 digits), which brings out the problem of big number addition. At present we assume that adding two n-digit numbers produces complexity of $O(n)$, with $O(\log n)$ methods displayed in Therefore, for $n > 200$, consider $T_{\frac{n}{2}} > 2^{0.694\times 100}$. Adding T_{n-1} and T_n takes $O(n)$ time. In total, T(n) takes $O(n^2)$ time.

Algorithm 2: fib2(n)

```
1 create an array f[0... n]
2 f[0] = 0, f[1] = 1
3 forall $i \geq 2$ do
4     $f[i] = f[i-1] + f[i-2]$
5 end
```

1.2.3 Matrix Multiplication and Fast Power Algorithm

We can rewrite (1) as:

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix},$$

which implies

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Therefore, it only remains to compute T^n for

$$T = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, T_n = \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix}$$

. Let $M(n), A(n)$ be the time complexity for computing the multiplication and addition of two n-bit integers respectively. Notice that $M(n) \geq n$, as we at least need to read the inputs. We have seen $A(n) = O(n)$. Therefore, $M(n) = \Omega(A(n))$.

We will design an algorithm to compute T^n and analyze the time complexity in terms of $M(n)$.

Let $k = \lceil \log_2 n \rceil$. We will compute all the $k+1$ matrices T^{2^0}, T^{2^1}, ..., T^{2^k}. Let $a_k a_{k-1} \cdots a_1 a_0$ be the binary representation of k. It is easy to see that

$$T^n = \prod_{i:a_i=1}^{k} T^{2^i} \tag{2}$$

The algorithm for computing T^n is thus split into two steps:

1. Compute T^{2^i} for $i = 0, 1, \ldots, k$.
2. Compute the product by (2).

Time Complexity for Step 1. Notice that it requires only one matrix multiplication to get $T^{2^{(i+1)}}$ from T^{2^i} if we've computed T^{2^i}, namely, $T^{2^{i+1}} = (T^{2^i}) \times T^{2^i}$. Now we analyze the time complexity for this multiplication by investigating the length of the for integers in T^{2^i}.

Lemma 1.2. The length of the integers in T^{2^i} is at most $2^{i+1} - 1$.

Proof. For the base step, if $i = 0$, then $T^{2^0} = T^1 = T$. The elements in T are less than $2^1 - 1 = 1$. For the induction step, suppose the lemma holds for $i - 1 (i > 2)$, then $T^{2^i} = (T^{2^{i-1}}) \times T^{2^{i-1}}$. Since th length of product of m-digit number and n-digit number is less than $m+n-1$, the length of element t in T is less than $(2^i - 1) + (2^i - 1) - 1 = 2^{i+1} - 3 < 2^{i+1} - 1$. □

Since multiplying two 2×2 matrices requires 8 integer multiplications and 4 integer additions, and $M(n) = \Omega(A(n))$, computing $T^{2^{i+1}}$ from T^{2^i} has complexity bounded by $c \cdot M(2^{i+1} - 1) \leq c \cdot M(2^{i+1})$ for some constant $c > 0$. Therefore, the overall time complexity for Step 1 is

$$c \cdot (M(2^2) + M(2^3) + \ldots + M(2^{2k+1})) \tag{3}$$

Lemma 1.3. For $i = 2, 3, \ldots, k$, $M(2^{i+1}) \geq 2M(2^i)$

Proof. To see this intuitively (although the following argument is not rigorous), if otherwise, we will have

$$M(2^i) < 2M(2^{i-1}) < 4M(2^{i-2}) < \ldots < 2^i M(1) = 2^i,$$

which contradicts $M(n) \geq n$ as mentioned earlier. □

Therefore, we have

$$M(2^{k+1}) \geq 2^1 M(2^k) \geq 2^2 M(2^{k-1}) \cdots \geq 2^i M(2^{k-i+1}) \cdots \geq 2^{k-1} M(2^2)$$

By dividing the inequation by 2^{k-i}, and pluging k with $\lceil \log_2 n \rceil$, we have

$$M(2^i) \leq \frac{1}{2^{k-i}} M(2^{k+1}) \leq \frac{1}{2^{k-i}} M(2n)$$

the time complexity for (3) is

$$c \cdot (M(2^2) + M(2^3) + \ldots + M(2^{2k+1})) \leq c \cdot M(2n) \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^{k-1}}\right) = O(M(2n))$$

, where the last equality is due to the fact that $M(2n) \leq 4M(n)$ (even if we use naive grade school multiplication) and $1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2k-1} < 2$.

Time Complexity for Step 2. We will analyze the time complexity for computing

$$\prod_{i=0}^{k} S_k := T^{2^i}$$

We need k matrix multiplications for S_k. Suppose we have computed $S_i := T^{2^0} \times T^{2^1} \times \ldots \times T^{2^i}$. Obviously, each entry of S_i is dominated by T^{2^i+1}. Thus, computing $S_{i+1} = S_i \times T^{2^{i+1}}$ requires at most $c \cdot M(2^{i+2})$ time. The overall time complexity for Step 2 is also at most

$$c \cdot (M(2^2) + M(2^3) + \ldots + M(2^{2k+1})),$$

which, by previous analysis, is $O(M(n))$.

Overall Time Complexity: $O(M(n))$

Remark 1.1. An Intuitive way to calculate the overall time complexity is to draw analogy between

$$1 + 2 + 2^2 + \cdots + n < 1 + \cdots + 2^{\lceil \log_2 n \rceil} = 2^{\lceil \log_2 n \rceil + 1} - 1 < 2n$$

and

$$M(2^2) + M(2^3) + \cdots + M(2^{k+1}) \leq M(2^2) + \cdots + M(2n)$$

2 Divide and Conquer

2.1 Multiplication

2.1.1 Problem Description

For naive two n-digit multiplication with base 2, suppose n is 2's power.

$$xy = (a \times 2^{n/2} + b)(c \times 2^{n/2} + d) = ac \times 2^n + (ad + bc) \times 2^{n/2} + bd$$

$$1234 \times 5678 = (12 \times 100 + 34) \times (56 \times 100 + 78)$$

We only consider multiplication, ignoring multiplying by 100 (which can be done simply by shifting a few digits) or addition, which are linear compared to Multiplication. Eventually, we need $\log_2 n$ steps to divide into the base step (1-digit times 1-digit).With each step forward, we multiply to number of elements on the level by 4. The multiplication is done on the end base, so in total we have $4^{\log_n 2} = O(n^2)$ complexity.

2.1.2 Karatsuba algorithm

Rather than calculating ad, bc, ac, bd separately; we obtain $(ad + bc) = (a + b) \times (c + d) - ac - bd$. Therefore, instead of dividing it into 4 parts, we divide it into 3 groups, through 3 multiplication operations on $\frac{n}{2}$, we get $3^{\log_2 n} = n^{\log_2 3} = n^{1.6}$ Specially, if n is not 2's power, we add zero to max bits, which at most doubles the original length, $(2n)^1.6$. No adding Zero in real coding, since carrying

Algorithm 3: Karatsuba algorithm

```
1 Input:Two n-digit numbers $x, y$.
2 Output: their product
3 $x_L, x_R = \text{leftmost}\lceil n/2 \rceil, textrightmost\lfloor n/2 \rfloor bitsofx$
4 $y_L, y_R = \text{leftmost}\lceil n/2 \rceil, textrightmost\lfloor n/2 \rfloor bitsofy$
5 P1=multiply($x_L, y_L$)
6 P2=multiply($x_R, y_R$)
7 P3=multiply($x_L + x_R, y_L + y_R$)
8 return $P1 \times 2^n + (P3 - P1 - P2) \times 2^{\frac{n}{2}} + P2$
```

occurs, (eg. 64-(32,(32+1)-¿64)). Time complexity is:

$$T(n) = 3T(\frac{n}{2}) + O(n) = 3T(\frac{n}{2}) + cn$$

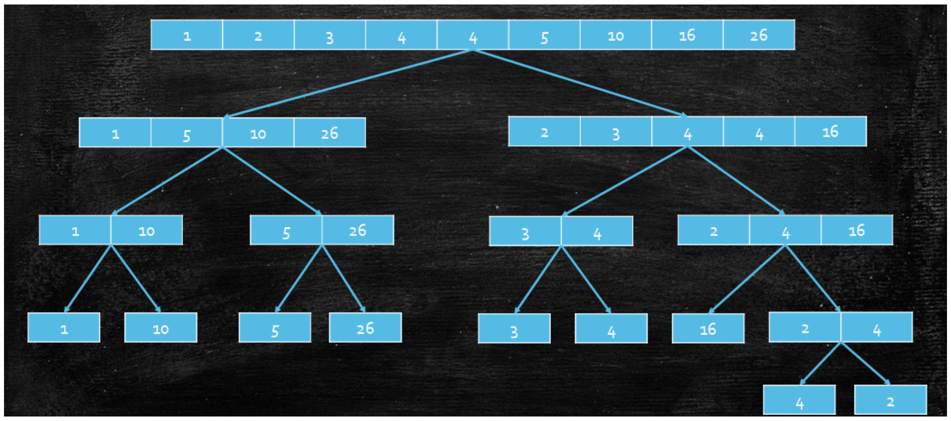


Figure 1: Merge Sort Pipeline

Due to carry, $T(\frac{n}{2})$ should be $T(\frac{n}{2}+1) = T(\frac{n}{2}) + O(n)$.

$$\begin{aligned} T(n) &= 3 \cdot (3T(\frac{n}{4}) + \frac{cn}{2}) + cn \\ &= 3^{\log_2 3} T(1) + cn(1 + \frac{3}{2} + \frac{3^2}{2^2} + \cdots + \frac{3^{\log_2 n}}{2^{\log_2 n}}) \\ &= O(n^{\log_2 3}) + O(n^{1+\log_2 1.5}) \\ &= O(n^{\log_2 3}) = O(n^{1.6}) \end{aligned}$$

1. **Toom-Cook**:$O(n^{1.465})$
   Breaking into 3 parts $5 \times \frac{n}{3} \times \frac{n}{3}$
2. **Fast-Fourier Transform**
3. **Strassen's magical idea for Matrix Multiplication**
   Squeeze eight multiplications into seven with complex calculation, better than the naive case (n for a row times a column, with n^2 numbers, in total $O(n^3)$. By normal divide and conquer, we divide one'n-size multi' into 8 $\frac{n}{2}$ multi, with total operations of $8^{\log_2 n} = O(n^3)$)

Remark 2.1. **Divide and Conquer** is a general algorithm design paradigm.

1. **Divide** Divide the problem into small size subproblems.
2. **Recursive** Solve small problems recursively
3. **combine** Combine the output of small size subproblems to get the answer for the original problem.
4. **Basic solver** If the problem size is small enough, solve it directly.

2.2 Sorting Problem

A very familiar problem, which can be easily picked up by analyzing pipeline1.

2.2.1 Time complexity

For the 'merge' part, merging two arrays(length m and n) requires $O(m+n)$ time. For each level, we have $O(n)$ merging operations. There are $\log_2 n$ levels, so the total time complexity is $O(n \log_2 n)$

Remark 2.2. Never use asymptomatic notations in an induction-based analysis!
An example of a wrong analysis is as follows:

Proof. $T(i) = O(i)$ holds for $i = 1, \dots, n$
Base Step: T1 =O(1) holds trivially.
Inductive Step: Suppose $T_i = O(i)$ holds for $i = 1, \dots, n-1$
$T(n) = 2T(\frac{n}{2}) + O(n) = 2O(\frac{n}{2}) + O(n) = O(n)$ □

There are two mistakes:

Firstly, the Inductive Step is meaningless, since $O(\cdot)$ takes in a function (with variable i). Given a specific value, $O(i = n-1) = O(1)$

Secondly, when the recursion happens numerous times, a constant can be a function of n. When we write $O(n)$, what we actually mean is cn.

Therefore, the correct proof should be:

Proof. $T(i) \leq ci$ holds for $i = 1, \dots, n$
Base Step: $T(1) \leq c$ holds trivially.
Inductive Step: Suppose $T(i) \leq ci$ holds for $i = 1, \dots, n-1$

$$\begin{aligned} T(n) = 2T(\frac{n}{2}) + O(n) &\leq 2T(\frac{n}{2}) + d \cdot n (\text{for some constant d}) \\ &\leq 2 \cdot c \cdot \frac{n}{2} + d \cdot n (\text{ by induction hypothesis}) \\ &= (c+d) \cdot n \end{aligned}$$

Induction fails as constant changes from c to $c+d$. □

A correct proof should be:

Proof. If $T(n) \leq 2T(\frac{n}{2}) + c \cdot n$ for constant c, $T(n) \leq B \cdot n \log n$ for some constant $B > c$.
Base Step: $T(2) \leq B \cdot 2 \log 2$ holds trivially.
Inductive Step: Suppose $T(i) \leq Bi \log i$ holds for $i = 2, \dots, n-1$
For $i = n$, we have

$$T(n) \leq 2T(\frac{n}{2}) + cn \leq 2 \cdot B \cdot \frac{n}{2} \log \frac{n}{2} + cn = Bn \log n - Bn + cn < Bn \log n$$

□

Remark 2.3. At least $\Omega(n \log n)$ comparisons are needed. // The time complexity is largely contributed by the running time of comparison. Total possible outputs is $3^{K(n)}$, n! numbers of permutations. Therefore, $3^{K(n)} \geq n!$, $K(n) \geq \log_3 n! \geq \Omega(n \log n)$

2.3 Inversions

Definition 2.1. if $a_i > a_j$ when $i < j$, we call (a_i, a_j) an inversion.

The idea is that we divide the input into two subsets (A:$x_1, x_2, \ldots, x_m$; B:$x_{m+1}, x_{m+2}, \ldots, x_{m+n-1}$), and count the inversions of each subset and **across** subsets, which takes $O(mn)$ with each a_i scanning the whole list B.

A magic trick is that we **mix merging and counting together**. We first count the inversions in A and B, then sort them separately. If A,B are sorted, we change a 'comparison' complexity problem into a 'search' complexity problem. In other words, we insert the additional step of counting inversion in the mergesort, by following operations, with time complexity of O(nlogn).

```
Algorithm 4: countInversions
Input: A list of n integers: S
Output: Number of Inversions: count
count=0
Divide the input into two subsets: A:$x_1, x_2, \ldots, x_{n/2}$; B:$x_{n/2+1}, x_{n/2+2}, , x_n$
if $n == 1$ then
    return 0
end
if $n == 2$ then
    if $x_1 < x_2$ then
        swap $x_1, x_2$
        return 1
    end
    else
        return 0
    end
end
count+=countInversions(A)+countInversions(B)
Maintain 2 pointers $i = 1, j = n/2 + 1$
while $i < n/2$ and $j < n$ do
    if $x_i < x_j$ then
        $i+ = 1$
    end
    else
        $j+ = 1$
        $count+ = j - n/2$
    end
end
count+=(n/2-i)×(j-n/2)
return count
```

2.4 Master Theorem

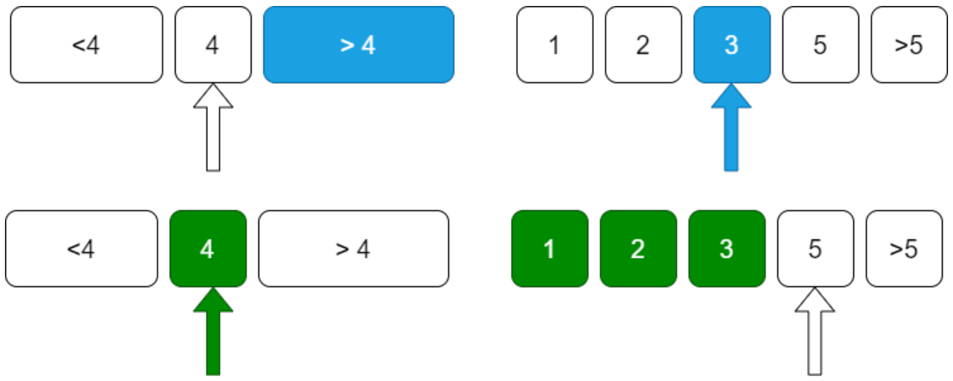


Figure 2: Inversion

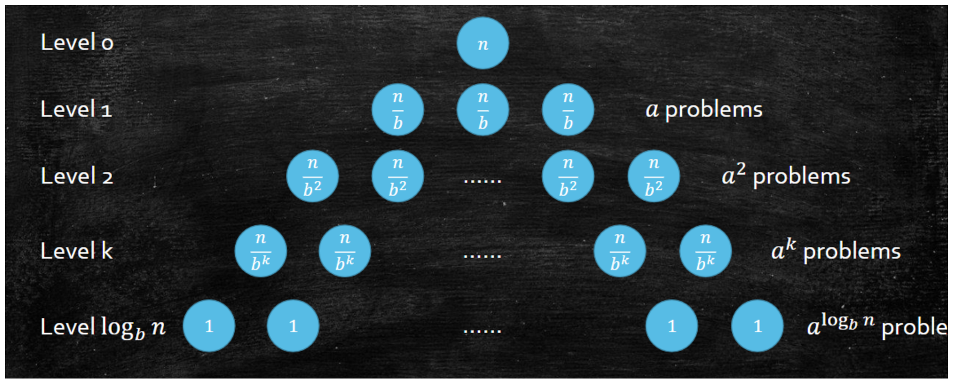


Figure 3: Master Theorem

Theorem 2.1. If $T(n) = aT(\frac{n}{b}) + O(n^d)$.
a:Divide into a subproblems
b:subproblem size n/b
$O(n^d)$:combining time complexity

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \\ O(n^d \log n) & \text{if } a = b^d \end{cases} \tag{4}$$

Proof. The running time of solving size-1 problem is $O(a^{\log_b n}) = O(n^{\log_b a})$. The total running time for combining is $c\dot(n^d) + c\dot((\frac{n}{b})^d)\dot a + \cdots a^k\dot c\dot(\frac{n}{b^k})^d + \cdots a^{\log_b n}\dot c$. The size-1 problem complexity is eaten up by the last element of the combining complexity. Eventually, we have $c\dot n^d\dot(1 + \frac{a}{b^d}) + \cdots (\frac{a}{b^d})^k + \cdots (\frac{a}{b^d})^{\log_b n}$. When $a < b^d$, intuitively the head is heavier(longer when imaging the operation of every layer as a line). When $a > b^d$, intuitively the tail is heavier. When $a = b^d$, $T(n) = O(n^d)\dot(1 \times \log_b n) = O(n^d \log_b n)$.

$\square$

2.5 Selection

Input: A set S of n integers $x_1, x_2, \dots, x_n$ and an integer k.
Output: The k-th smallest integer x^* among $x_1, x_2, \dots, x_n$.
naive case: Sorting and choosing the k-th element: O(nlogn)
alg 1: quick sort? have the first number to be the pivot.
alg 2: Divide $x_1, x_2, \dots, x_n$ into three subsets, recursively pick out the subset x^* is in.

Algorithm 5: Select

```
1  Choose an **arbitrary** value $v$ among $x_1, x_2, \dots,$
2  Divide $x_1, x_2, \dots, x_n$ into three subsets $L, M, R$
3  if $k \leq |L|$ then
4      return Select(L,k)
5  end
6  else if $|L| < k \leq |L| + |M|$ then
7      return $v$
8  end
9  else if $|L| + |M| < k$ then
10     return Select(R,$k - |L| - |M|$)
11 end
```

Proof. **the scale of the problem shrinks:**
First, we show that the algorithm always terminates. If it does not terminate in the current recursive iteration, it will either call Select(L,k) or Select(R, $k - L - |M|$). It suffices to show that $|L|$ ¡ n and $|R|$¡ n (so that the problem size is strictly decreasing). This is guaranteed by $v \notin L$ and $v \notin R$ (v is in the set, so at least v is excluded).
the algrithm is correct inductively:
Suppose the correct value is output for any array S of length n and any $k \in 1, 2, \dots, n$. **Base Step:** n = k = 1, which is obviously true.
Inductive Step: Suppose the algorithm correctly outputs the k-th smallest value for any array S with any length $l \in 1, 2, \dots, n-1$ and any $k \in 1, 2, \dots, n$.
Select(L,k) returns the k-th smallest value in L and Select(R, $k - |L| - |M|$) returns the $(k - |L| - |M|)$-th smallest value in R by induction hypothesis. For any input array of length n and any integer v in it, it is straightforward to check that the k-th smallest value in L is the k-th smallest value in S if $L \geq k$, and the remaining two cases are also correct.

□

2.5.1 Time Complexity

For dividion,$\Theta(n)$ with each element compared to v. For recursion, $T(|L|)$ or $T(|R|)$ or $O(1)$.
For the worst case, consider $1, 2, ..., n$,where k=n. The algorithm is $O(n^2)$.

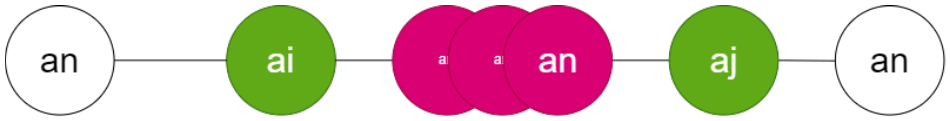


Figure 4: The white an are doesn't matter terms. We only consider the last step when $a_m, m \in [i, j]$ will be chosen as a pivot. There are (j-i+1) choices, and only 2 choices (namely, a_i or a_j) can result in comparison between a_i and a_j.

Remark 2.4. Randomness in input and randomness in algorithm
Commonly, we choose to analyze the mean time complexity of a random algorithm. Because in real case, randomness of input is a strong condition(block-like features commonly).

By estimation, for every layer, n comparisons. If we are lucky enough to have $\frac{1}{3}$ in the minimum separation, then $n\frac{2}{3}^n > 1$, $k < \log_1 .5n$.

Definition 2.2. $E[x] = \sum_{\omega \in \Omega}^{n} \Pr(\omega) \cdot X(\omega)$

Theorem 2.2. Linearity of expectation

$$E|\sum_{i=1}^{n} X_i| = \sum_{i=1}^{n} E[X_i]$$

holds when $X_i, \ldots X_n$ are dependent.

eg. For dices, $E[x] = \frac{1}{6} \cdot (1 + 2 + \cdots + 6)$. When rowing two dices, by the Linearity of expectation, $E[x] = 3.5 + 3.5 = 7$. If two dices output the same number magically, we still have $E(x) = (2 + 4 + 6 + ... + 12)/6 = 7$.

For our case, since comparison takes up most of the running time, we calculalte the average times of comparison. Firstly, if a_i, a_j are compared, one must be pivot and the other goes to the next layer of the division tree, which will only be compared once; When we consider the ordered sequence, a_i, a_j are compared iff no element betweeen them are chosen as pivots, else they will be in different subsets, as is shown in 4.

$$E[X_{ij}] = \Pr(X_{ij} = 1) \cdot 1 + \Pr(X_{ij} = 0) \cdot 0 = Pr(X_{ij} = 1) = \frac{2}{j - i + 1}$$

$$i = 1 : \frac{2}{2} + \frac{2}{3} + \cdots + \frac{2}{n} = \sum_{i=1}^{n} \frac{2}{i} = 2 \cdot \sum_{i=1}^{n} \frac{1}{i} = 2 \cdot \log n$$

$$i = 2 : \frac{2}{2} + \frac{2}{3} + \cdots + \frac{2}{n-1} \ldots$$

To sum up, worst-case $O(n^2)$, expected average running time $\Theta(n \log n)$

2.5.2 Median of the medians(1973)

How to pick a **good pivot**? In other words, how to make sure $|L|$ and $|R|$ are approximately equal?

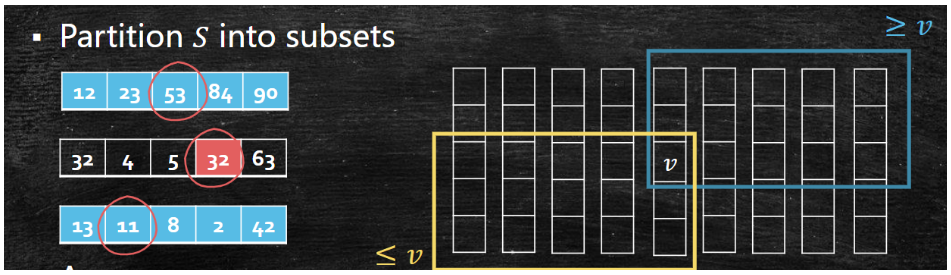


Figure 5: Median of the medians

Partition S into subsets with size 5. O(n) Find the medians of each subset: $O(n/5)$ (since operations in a fixed-sized set is O(1)).
Find the median of the medians: $T(n/5)$.
Elements remaining: According to 5, there is $n - \frac{n}{5} \cdot \frac{1}{2} \cdot 3 = \frac{7n}{10}$ elements left.

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

Proof. $T(n) = O(n)$
Assume that $T(n) \leq cn$
$T(\frac{n}{5}) < \frac{cn}{5}$,$T(\frac{7n}{10}) < \frac{7cn}{10}$, $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n) < 0.9cn + bn \leq cn$. which holds when we set c to be bigger then b.

□

If the size of subset is i=2m+1, $m \in N$. To obtain the median of medians, we have $O(\frac{n}{2m+1})$. There are $n - \frac{n}{2m+1} \cdot \frac{1}{2} \cdot (m+1) = \frac{(3m+1)n}{2(2m+1)}$ elements left.

$$T(n) = T(\frac{n}{2m+1}) + T(\frac{(3m+1)n}{2(2m+1)}) + O(n) = T(\frac{(3m+3)n}{2(2m+1)})$$

However, in real case, the const is much bigger than 'quick sort' approach with a random pivot.

2.6 Clostest Pair

We can improve the naive algorithm by **sorting**. Imagine in 1-dim case, sorting ($O(n \log n)$) plus comparison between neighbouring points ($O(n)$) results in an $O(n \log n)$ algorithm.

Consider 2-dim case, apparently using only x sequence or y sequence cannot lead us to the right answer. Some tricks are implemented:

1. **First sort by x, then by y**
   The only use of sorting by x is to divide the original dots into x-neighbouring subsets. After the division, any changes inside the subsets are allowed, so we merge the sort by y-axis step into the conquer step, which eliminates the $O(n \log n)$ sorting operation in each layer.

2. **Bound the search plane**
   If we've found the minimum distance inside A and B separately, we only need to consider the distance between points across A and B. Additionally, $|x_i - x_{mid}| < dist$,which forms the $2 - \delta strip$ shown in fig. Similarly, during the conquer step, for each point p_i, we only consider p_j that satisfies $|y_i - y_j| < dist$. By proposition 2.1, we magically found that only the distance between **four** neighbouring points are considered for each point (as is shown in fig 7)

Proposition 2.1. At most 4 points can appear in a $\delta \times \delta$ square, so that the distance between any two points are no smaller than δ.

Proof. Divide the square into four smaller squares, as in shown in fig 6. Since two points are at most $\frac{\delta}{\sqrt{2}} < \delta$ apart, at most one point can exist in each small square. So in total, at most four points appear in the square. □

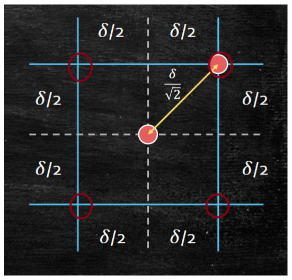


Figure 6: At most 4 points appear in the square

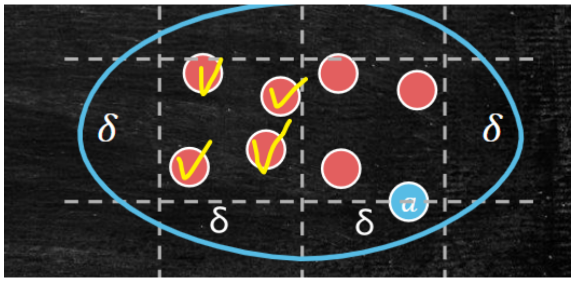


Figure 7: Only four neighbouring points need to be considered for each point a

This brings out an $O(n \log n)$ alogorithm. The sorting by x-coordinate takes $O(n \log n)$, and the 'divide and conquer' step amounts to adding constant steps(since each point only has four neighbouring points to count) to MergeSort.

Algorithm 6: FindClosestPair

```
Input: A list of pairs P: $p_1 = (a_1, b_1), \dots p_i = (a_i, b_i), \dots, p_n = (a_n, b_n)$
Output: distance of closest two pairs in P: dist
if $|P| == 1$ then
    return
end
Sort P by x-coordinate.
Divide P into two equally sized subsets A,B, the point in the middle $p_{mid} = (a_{mid}, b_{mid})$.
dist=min(FindClosestPair(A),FindClosestPair(B))
Merge A and B by y-cordinate to form C.
foreach $c_i = (a_i, b_i)$ in $C$ do
    if $|a_i - a_{mid}| < dist$ then
        add $c_i$ to S
    end
end
foreach $s_i = (a_i, b_i)$ in $S$ do
    j=i+1
    while $|b_i - b_j| < dist$ do
        dist=min(dist, $\|b_i - b_{i+1}\|$)
        j=j+1
    end
end
return dist
```

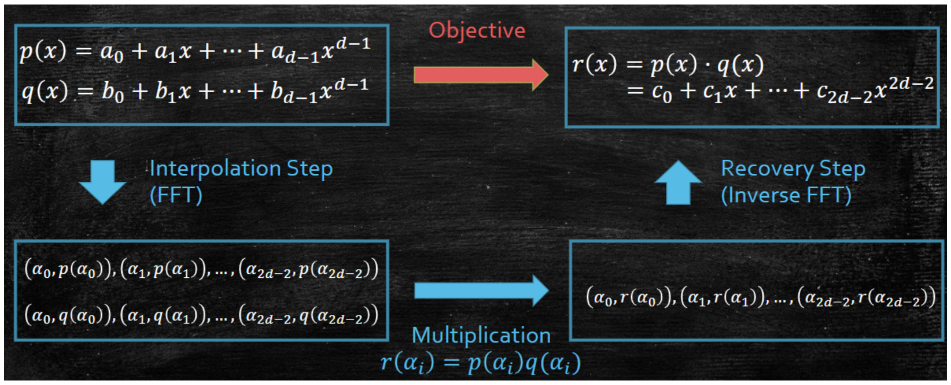

$p(x) = a_0 + a_1 x + \dots + a_{d-1}x^{d-1}$
$q(x) = b_0 + b_1 x + \dots + b_{d-1}x^{d-1}$
Objective
$r(x) = p(x) \cdot q(x) = c_0 + c_1 x + \dots + c_{2d-2}x^{2d-2}$
Interpolation Step (FFT)
Recovery Step (Inverse FFT)
$(\alpha_0, p(\alpha_0)), (\alpha_1, p(\alpha_1)), \dots, (\alpha_{2d-2}, p(\alpha_{2d-2}))$
$(\alpha_0, q(\alpha_0)), (\alpha_1, q(\alpha_1)), \dots, (\alpha_{2d-2}, q(\alpha_{2d-2}))$
Multiplication
$r(\alpha_i) = p(\alpha_i)q(\alpha_i)$
$(\alpha_0, r(\alpha_0)), (\alpha_1, r(\alpha_1)), \dots, (\alpha_{2d-2}, r(\alpha_{2d-2}))$

Figure 8: FFT

2.7 Fast Fourier Transform and Polynomial Multiplication

We use array $(a_1, a_2, \dots, a_{d-1})$ to describe $p(x) = \sum_{i=0}^{i=d-1} a_i x^i$

$$r(x) = \sum_{i=0}^{2d-2} c_i x^i, \text{where } c_i = \sum_{k=0}^{i} a_k b_{i-k}$$

By Karasuba Algorithm, we can have $O(n \log_{1.5} n)$ algorithm. The key of interpolation is to use the minimum number of points to describe the original subject. Any polynomial is a node in the n-dim space, with its coordinates $(a_1, a_2, \dots, a_{d-1})$. We need $2d-1$ interpolation points to fully store the answer polynomial by theorem .

Theorem 2.3.

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{d-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{d-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{d-1} & x_{d-1}^2 & \cdots & x_{d-1}^{d-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{d-1} \end{bmatrix}$$

Since the determinant of a Vandermonde matrix is $\Pi_{0 \le i < j \le d-1}(x_j - x_i)$. Denote the middle matrix as A, A is an invertible matrix.

interpolation step: For counting $p(a_i)$, we need d addition for (2d-1) interpolations, which brings out $O(n^2)$. However, we can try to compute $p(a_{i+1})$ from a_i by carefully chosen interpolation points. If we choose $a = \alpha$ and $a = -\alpha$, we can calculate $a_0 + a_2\alpha^2 + \dots$ and $a_1 + a_3^3 + \dots$ with d times, and reduce (2d) to obtain both $p(\alpha)$ and $p(-\alpha)$ with d times.

$T(D) = 2T(\frac{D}{2}) + O(n)$ is wrong, because the algorithm cannot be done recursively. In the second layer, $\alpha_0^2 = -\alpha_2^2$ cannot be satisfied. In the complex plane, however, for every element, there exists two roots.

multiplication of two complex numbers $r(a_i) - p(a_i)q(a_i)$
recovery: How to find A^{-1} and calculate matrix multiplication?

$$)\ ($$

If columns are orthonormal, so are the rows. $AA^* = E, A^{-1} = A^*$. Intuitively, the inverse of a orthonormal matrix can be obtainly easily.

Proposition 2.2. $\frac{1}{\sqrt{D}}A(w)$ is orthonormal for $w = e^{\frac{2\pi i}{D}}$

3 Graph

Terminologies

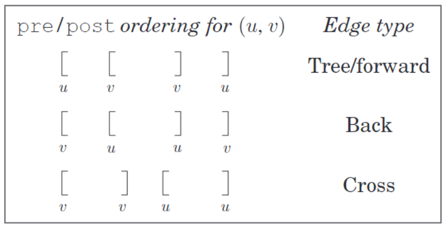
1. vertex, vertice, edge, edges, index, index
2. Adjacency Matrix, Adjacency List
3. reachability, connected component
4. Tree: a connected and acyclic graph, containing backedge and Tree edge.
5. DAG: Directed Acyclic Graph
6. Sink(Source): A collection of vertice that doesn't have outgoing(ingoing) edges.
7. Partition: $C_1, C_2, \ldots, C_n$ is a partition of C iff their union is C and they don't intersect.
8. cut: Given a partion A,B of P, value defined by $c(A, B) = |E(A, B)|$(number of edges across two sets).

3.1 Depth First Search

3.1.1 Existence of cycles

Use dfs to traverse all the vertice in a graph. First reach a vertex, then visit its connected component. Most effective when stored in an adjacency list. Dfs can form a tree(forest).
Specially, we analyze the case of directed graph. For directed graph, there exists 4 kinds of edges:

1. **Tree Edges** Edges in the DFS search: (u, v) where marked(v) = false
2. **Forward Edges** Edges that point to non-child descendants
3. **Back Edges** Edges that point to ancestors
4. **Cross Edges** All the other edges: edges point to vertices on other tree doesn't exist for undirected graph, since if (c, f) occurs it will only be tree edge.



pre/post *ordering for* (u, v)	*Edge type*
$[_u \; [_v \;]_v \;]_u$	Tree/forward
$[_v \; [_u \;]_u \;]_v$	Back
$[_v \;]_v \; [_u \;]_u$	Cross

Figure 9: Directed Graph

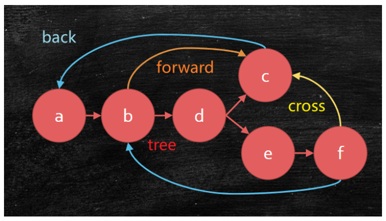


Figure 10: colored edges

Correctness of Algorithm:
If there exists a back edge in directed graph, a cycle exists.
If a cycle exists, then a backedge must be found. Since a tree is acyclic.
By algorithm 7 plus the condition of the existence of a backedge ($pre[v] < pre[u]$ and $post[u] < post[v]$), depicted in 9

Algorithm 7: DFS on Directed Graph

```
clk=0
foreach $v \in V$ do
    if marked[v]=False then
        explore($v$)
        clk++ # used to calculate the number of connected components
    end
end
Function explore($u$):
    pre[u]=clk
    clk++
    visited[u]=false
    foreach $(u, v) \in E$ do
        if marked[v]=false then
            explore ($v$)
        end
    end
    post[u]=clk
    clk++
```

3.1.2 Application: Topological Ordering

A DAG must have topological ordering, since:

1. Every DAG have a sink.
   Intuitively, if we start at v, since there exists an outgoing edge for every vertex. Since DAG doesn't have cycles, in finite steps, we'll visit every vertex,

2. Find a topological order.
   When deleting sink vertice and edges pointing to them, we obtain a new DAG. By repeting the operation of finding sink (visiting sink at last), and deleting sink to construct new graph, and finding sink in the new graph... We successfully find a topological ordering by following the inverse sequence of the sinks.

Time Complexity:
Finding a sink takes $O(|V|)$, removing a sink to update graph with $|V|$ rounds.
A trick is that the vertice with the earliest post ordering is a sink. If not, the outgoing point points to another vertex that finishes earlier. (consider the fact that no edge (u,v) exists in DAG that satisfied $post[v] > post[u]$. See 9, backedges satisfy, but doesn't exist in DAG) Therefore, the topological ordering is the inverse order of finishing points, which brings the time complexity of DFS down to $|E| + |V|$.
Correctness:
If (y,x) exists, y is not a sink since x is in the graph, contradiction!

3.1.3 Strongly Connected Components for Directed Graph

All the SCC forms a partition of C. i.e. for every v, $\exists C_i$ so that $v \in C_i$. For completeness, by construction, every single v can be formed as a SCC. For non-intersection, if $v \in C_i, C_j$, C_i and C_j together forms a bigger C.C., as every vertex from C_i can reach points from C_j through v.

- Note that we can't simply employ DFS to find C.C., as although you find reachable points from a certain vertice v, the vertice you explore may not be reachable from each other.
- Note that unlike topological ordering, the node with the earliest post time doesn't necessarily occur in sink SCC, as the vertice with outer edges are previsited where our algorithm ends in somewhere else.i.e.,for graph 11 consider DFS in sequence 5,6,7,8, apparently, the 8 has the smallest post value, but is not in sink.
- However, the vertice with biggest post time must be in source SCC. Assume u has the largest finish time, which must be a root of one DFS tree. Suppose u isn't in source SCC, i.e. $\exists v$ s.t. (v, u) exists. v isn't in the tree of u, as (u, v) doesn't exist. ,v cannot start earlier than u, v isn't in another DFS tree.
- Consider the meta-graph, where connected components form a big node, it must be DAG, else contradict with biggest SCC, therefore, there must be a sink. Also, a way to find whether v is reachable from u is to check the connectivity of bigger SCCs, U and V.

Algorithm 8: Strongly Connected Component

```
Input: DAG:G
Output: num of SCC
1 Deduce G's inverse $G^R$ from G
2 Run DFS on $G^R$ to find source SSC, which is sink SSC of $G$, record it in the descending finish
  time as F.
3 DFS on $G$ in the order of F.
```

Time Complexity: Naive Case: Find a sink, delete it and repeat again...which takes $O(|V||E|)$. Better Case: Doing DFS twice, shown in algorithm 8, taking $O(|V| + |E|)$.

3.2 Breadth First Search

3.2.1 Unweighed Shortest Path in Undirected Graph

Algorithm 9: alg:bfs

```
Queue Q;
Function bfs(,$Q$,$u$):
    for each $v \in V$ $marked[v] \leftarrow false$
    Q.push(s) $marked[s] \leftarrow true$ while $Q$ is not empty do
        u=Q.top();Q.pop();
        foreach $(u, v) \in E$ do
            if marked[v]=false then
                marked[v]=True
                Q.push(v)
            end
        end
    end
```

Consider doing topological order in BFS. Firstly it cannot tell the difference between crossedge and backedge, and it has no forward edge(as the only chance of visiting unvisited nodes forms a tree edge , i.e. nodes in layer 2 can only reach unvisited nodes and index them with 3). Additionally, DFS cannot detect cycles(=detect backedge), or find SCCs(based on cycles).

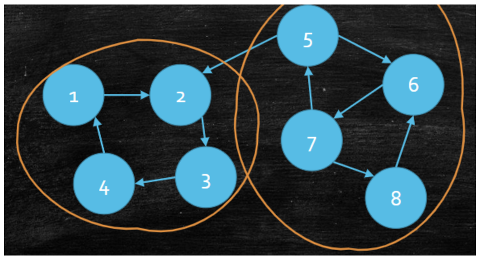


Figure 11: SCC

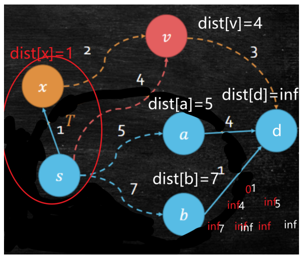


Figure 12: Correctness of Dijkstra's algorithm

3.2.2 Dijkstra: SP. with positive weight

Remark 3.1. difference between quantity and numerical value
Pay attention to the length of a numerical value. i.e., input length is n, the actual number complexity. Storing n quantities takes n storage space, but storing a n-length numerical value takes $\log n$ bits. Therefore, an increase in bits takes exponencial time complexity. $O(|V| + |E|)$, |'| is a quantity $O(w_max)$ is a numerical value.

- Works by keep dragging the closest vertex to the current SPT and updating the minimum path length.
- Dijkstra algorithm is based on the assumption that evrey step out of SPT(shortest path tree) at least increases the path length, so it only solves graphs without negative cycles.
- A trick is to use heap (binary-heap, d-heap, Fibonachi-heap) to store the shortest distance from other vertice to s, so that we can quickly obtain the closest node from SPT, without the need to traverse all outer edges again. Creating the heap takes $O(|V| \log |V|)$, picking the top vertex takes $O(\log |V|)$, and updating all the nodes takes $O((|V| + |E|) \log |V|)$. i.e, for figure ??, consider the first step. We update x,a,b,v and the closest vertex x natually pops to the top.

Correctness of algorithm: If there exists $x \notin SPT, v \in SPT$, s.t. $dist(s \to x \to v) < dist(v)$, $dist(x) < dist(v)$, x should be in SPT,contradiction!

3.2.3 Bellman-Ford: shortest path with negative weight

Dijkstra's algorithm works because the dist value it maintains is either overestimates or exactly correct. The key idea of Bellman-Ford is that it updates all the edges (kind of 'edgewise' operation) each time (different from Dijkstra's algorithm who updates nodes one step connected to SPT), which takes $O(|V||E|)$.

Correctness of algorithm:

1. After k rounds, dist(v) is the shortest path of all k-edge-paths.($dist[u_k] \leq d(u_1, u_2, \ldots, u_k)$) For base step, dist[s] is the shortest dist of all 0-edge-paths.
   For induction step, suppose it is true for $k-1$ rounds, $dist[u_{k-1}] \leq d(u_1, \ldots, u_{k-1})$. By Bellman-ford update condition, $dist[u_k] \leq d(u_1, \ldots, u_{k-1}) + w(u_{k-1}, v)$, the righthand side means all the other k-edge-paths.
2. For graphs without negative cycle, all the shortest paths will have at most $|V| - 1$ edges. If there are more than $|V|$ paths, a vertex must be visited twice, but by erasing the whole cycle results in a shorter path, contradiction!
3. Negative cycle exists iff some dist is updated in the last($|V|^{th}$) round.

Algorithm 10: Di(j)kstra

```
Input: G=(V,E),s
Output: dist(u) is the shortest distance from s to u
# initialize foreach $u \in V$ do
    dist(u)=inf
    pre(u)=Nan
end
dist(s)=0
H=makequeue(V)
while H is not empty do
    u=H.top();H.pop()
    foreach $(u, v) \in E$ do
        if $dist(v) > dist(u) + l(u, v)$ then
            $dist(v) = dist(u) + l(u, v)$ # update the shortest dist
            pre(v)=u# record previous node
        end
    end
end
```

```
Input: G=(V,E),s
dist[s]=0, dist[x]=inf for vertex other than s
pre(v)=Nan
while some dist[x] updates do
    # update at most —V— rounds. foreach $(u, v) \in E$ do
        if dist[v]¿dist[u]+w(u,v) then
            dist[v]=dist[u]+w(u,v)
            pre(v)=u
        end
    end
end
```

4. If no vertice updates in V^{th} round, then no vertice will be updated then. But if a single vertice is not updated.
5. If a vertex is upgraded after V rounds, then its path must include a negative cycle. A negative cycle can be found by tracing the recorded shortest path until two repeated nodes occurs. Tricks include running till the $2|V|^{th}$ round and see the vertice upgraded in $|V|^{th}$ and $2|V|^th$.