Notes for Algorithms: Design and Analysis

Jing LI
pkuyplijing@gmail.com

July 9, 2016

Sincere gratitude to Professor Tim Roughgarden
for offering such a wonderful class.

Contents

Chapter 1

Introduction

1.1 Warmup: Integer Multiplication Problem

1.1.1 The Primary School Approach

Consider the integer multiplication algorithm that everyone learned in primary school.

Input two n-digit numbers x and y.

Output the product $x \times y$.

We will assess its performance by counting the number of **primitive operations**, here being the addition or multiplication of two single-digit numbers, required to carry it out. In this case, it is clearly $\Theta(n^2)$. It might have been taken for granted that this is the unique, or at least optimal approach, while actually it isn't. As Aho, Hopcroft and Ullman put it in their 1974 book *The Design and Analysis of Computer Algorithms*, "Perhaps the most important principle for the good algorithm designer is to refuse to be content." Always be ready to ask yourself the question: CAN WE DO BETTER?

1.1.2 Karatsuba Multiplication

Consider the calculation of 5678×1234. We will note $a = 56, b = 78, c = 12$ and $d = 34$. The calculation can be carried out with the following steps:

1. Calculate $a \cdot c = 672$;
2. Calculate $b \cdot d = 2652$;
3. Calculate $(a + b)(c + d) = 134 \times 46 = 6164$;
4. Calculate ③ - ② - ① = 2840;
5. Calculate ①×10000 + ④×100 + ② = 6720000 + 284000 + 2652 = 7006652.

The procedure here can be formalized into a recursive algorithm to calculate $x \times y$. Write $x = 10^{n/2}a + b$ and $y = 10^{n/2}c + d$. Obviously we have $xy = 10^n ac + 10^{n/2}(ad + bc) + bd = 10^n ac + 10^{n/2}((a+b)(c+d) - ac - bd) + bd$, which inspires us of the following algorithm:

Algorithm 1.1 Karatsuba Multiplication

1: Recursively calculate ac;
2: Recursively calculate bc
3: Recursively calculate $(a + b)(c + d)$
4: Calculate ③ - ② - ① to get $ad + bc$
5: Combine the results appropriately, i.e. $xy = 10^n ac + 10^{n/2}(ad + bc) + bd$

Algorithm 1.1 demonstrates that even a simple problem with a presumably unique solution has plenty of room for subtle algorithm analysis and design.

1.2 Course Topic

The course will cover the following topics:

- Vocabulary for design and analysis of algorithms;
- Divide-and-conquer algorithm design paradigm;
- Randomization in algorithm design;
- Primitives for reasoning about graphs;
- Use and implementation of data structures.

In part II of the course the following topics will be covered:

- Greedy algorithm design paradigm;
- Dynamic programming algorithm design paradigm;
- NP-complete problems and what to do with them;
- Fast heuristics with provable guarantees for NP problems;
- Fast exact algorithms for special cases of NP problems;
- Exact algorithms that beat brute-force search for NP problems.

1.3 Merge Sort

We will use **merge sort** to illustrate a few basic ideas of the course. Merge sort is a non-trivial algorithm to tackle the sorting problem:

Input An unsorted array of n numbers;

Output The same numbers in sorted order.

1.3.1 Pseudo Code

Merge sort is more efficient than selection sort, insertion sort and bubble sort. It is a good introductory example for the divide & conquer paradigm. Its pseudo code is shown in Algorithm 1.2. The merging process may not seem intuitive.

Algorithm 1.2 Merge sort

input:
Unsorted array of length n
output:
Sorted array of length n
1: **if** length of the array = 0 or 1 **then**
2: return ▷ Basic case. Array already sorted.
3: Recursively merge sort 1st half of the array.
4: Recursively merge sort 2nd half of the array.
5: Merge the two sorted halves into one sorted array.

It is implemented with parallel traverses of the two sorted sub-arrays, as shown in Algorithm 1.3.

Algorithm 1.3 Merging two sorted sub-arrays

input:
A = 1st sorted sub-array, of length $\lfloor n/2 \rfloor$
B = 2nd sorted sub-array, of length $\lceil n/2 \rceil$
output:
C = sorted array of length n
1: i = 1, j = 1
2: **for** k = 1 **to** n **do**
3: **if** i > A.len **then** ▷ A has been exhausted
4: C[k] = B[j++]
5: **else if** j > B.len **then** ▷ B has been exhausted
6: C[k] = A[i++]
7: **else if** A[i] < B[j] **then**
8: C[k] = A[i++]
9: **else** ▷ A[i]≥B[j]
10: C[k] = B[j++]

1.3.2 Running Time

The running time of the merging operation is obviously linear to the length of the array. Precisely speaking, each iteration involves one increment of i or j, one increment of k, an assignment to C[k] and at most 3 comparisons[1]. Taking

[1]Here we are taking an approach more detailed than that in the lecture: end cases, i.e. when A or B is exhausted, are taken into account.

the initialization of i and j into account, in total we have to carry out 6n + 2 primitive operations, which is smaller than 8n.

We can then draw the **recursive tree** of the problem. The original merge sort problem of size n resides at level 0. At level 1 we have 2 sub-problems of size $n/2$, etc. In general, at level j we have 2^j sub-problems of size $\frac{n}{2^j}$, and in total we have $\log n + 1$ levels[2]. At each level, the number of required primitive operations is smaller than $8 \cdot \frac{n}{2^j} \cdot 2^j = 8n$. In the end, we have an upper bound of the total number of primitive operations needed to solve the original merge sort problem of size n: $8n(\log n + 1)$.

1.4 Asymptotic Analysis

In the analysis above, we have been applying 3 guiding principles that will serve as universal tactics in future analysis of algorithms:

1. Focus on worst-case analysis, rather than average-case analysis or benchmarks on a specified set of inputs.
2. Analyze with no regard to the constant factor.
3. Conduct asymptotic analysis, i.e. focus on running time when n is large.

Asymptotic analysis provides basic vocabulary for the design and analysis of algorithms. It is essential for high-level reasoning about algorithms because it is both coarse enough to suppress details dependent upon architecture, language, compiler and implementation details, and sharp enough to facilitate comparisons between different algorithms, especially for inputs of large size. Its high-level idea is to **suppress constant factors as well as lower-order terms**. For our example of merge sort, the running time of $8n(\log n + 1)$ is actually equivalent to $n \log n$, or in big-O notation, $O(n \log n)$.

1.4.1 Big-O Notation

Let $T(n), n \in \mathbb{N}$ be the function representing the running time of an algorithm with input of size n. The **Big-O notation** $T(n) = O(f(n))$ means that eventually (for all sufficiently large n), $T(n)$ will be bounded above by a constant multiple of $f(n)$. We hereby provide its formal definition.

Definition 1. *Big-O notation $T(n) = O(f(n))$ holds if and only if there exist constants c, n_0 such that*

$$T(n) \leq c \cdot f(n), \forall n \geq n_0.$$

Theorem 1.

$$a_k n^k + \cdots + a_1 n + a_0 = O(n^k)$$

[2] At the last level k we must have $\frac{n}{2^k} = 1$, thus $k = \log n$.

Proof. Constants $n_0 = 1$ and $c = \sum_{i=0}^{k} |a_i|$ satisfy Definition 1. □

Theorem 2. *For every $k \geq 1$, $n^k \neq O(n^{k-1})$.*

Proof. The theorem can be proved by contradiction. Suppose $n^k = O(n^{k-1})$, i.e. $\exists$ constants c, n_0 such that

$$n^k \leq c \cdot n^{k-1}, \forall n > n_0.$$

Then

$$n \leq c, \forall n > n_0,$$

which is an obvious contradiction. □

1.4.2 Omega, Theta and Little-O Notations

Definition 2. *Omega notation $T(n) = \Omega(f(n))$ holds if and only if there exist constants c, n_0 such that*

$$T(n) \geq c \cdot f(n), n \geq n_0.$$

Definition 3. *Theta notation $T(n) = \Theta(f(n))$ holds if and only if $T(n) = \Omega(f(n))$ and $T(n) = O(f(n))$, which is equivalent to $\exists$ constants c_1, c_2, n_0 such that*

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n), \forall n \geq n_0$$

A convention in algorithm design, though a sloppy one, is to use big-O notation to represent Theta notation.

Definition 4. *Little-O notation $T(n) = o(f(n))$ holds if and only if $\forall$ constant $c > 0$, $\exists$ constant n_0 such that*

$$T(n) \leq c \cdot f(n), \forall n \geq n_0.$$

Theorem 3. $n^{k-1} = o(n^k), \forall k \geq 1$

Proof. Constant $n = 1/c$ can satisfy the condition. □

Chapter 2

Divide and Conquer Algorithms

A typical divide-and-conquer solution to a problem consists of the following steps:

1. Divide the problem into smaller sub-problems.
2. Conquer the sub-problems via recursive calls.
3. Combine solutions of sub-problems into a solution to the original problem, often involving some clean-up work.

2.1 Inversion Counting Problem

We will solve the inversion problem using the divide-and-conquer paradigm. The problem is described as follow.

Input An array A containing numbers $1, 2, \ldots, n$ in some arbitrary order.

Output Number of inversions in this array, i.e. number of pairs $[i, j]$ such that $i < j$ and $A[i] > A[j]$.

A geometrical solution to the problem is to draw two parallel series of points, mark one series in the order inside array A, and mark the other in the order $1, 2, \ldots, n$. Connect points marked by the same number, i.e. 1 with 1, 2 with 2, etc, then the number of crossing lines is exactly the number of inversions.

The inversion number is widely useful in comparison and recommendation systems. A movie rating website wants to compare tastes of its users and recommend to a user movies liked by other users with similar taste to his. One criterion of such comparison is to pick the ratings given by one user to a series of movies and compare them against other users' ratings by calculating the number of inversions. The fewer inversions there are, the more similar their tastes are.

A brute-force approach is obviously $\Theta(n^2)$. We can do better by applying the divide-and-conquer paradigm. Suppose that the array has been divided into two halves. An inversion $[i, j]$ is called a left inversion if both i, j are in the left half, a right inversion if they are both in the right half, and a split inversion if i is in the left half and j is in the right half. A high-level divide-and-conquer algorithm is provided in Algorithm 2.1. If `countSplitInv` can be implemented as $\Theta(n)$, then the whole algorithm will be $\Theta(n \log n)$.

Algorithm 2.1 Divide-and-conquer Inversion Counting

input:
Array A
output:
Number of inversions in A

```
1: function COUNT(Array A)
2:     if A.len == 1 then
3:         return 0
4:     else
5:         x = count(1st half of A)          ▷ Left inversions.
6:         y = count(2nd half of A)          ▷ Right inversions.
7:         z = countSplitInv(A)              ▷ Count split inversions.
8:         return x+y+z
```

The implementation of `countSplitInv` seems quite subtle, but it can actually be developed from merge sort. In Algorithm 2.1, the subroutine `count` only counts the number of inversions. In addition to that, we rename it with `sortAndCount` and require that it also sorts the array. Subroutine `countSplitInv` now becomes `mergeAndCountSplitInv`. It merges the two sorted sub-arrays into one sorted array and counts the number of split inversions.

If there exists no split inversion, it must be the case that any element of the left sub-array A is smaller than any element of the right sub-array B. As a result, when merging the two sorted sub-arrays, A will be exhausted before any element of B is put in the result. Once an element of B is chosen during the merging process before A is exhausted, every element left in A forms an inversion with it. Algorithm 2.2, developed from Algorithm 1.3, uses this idea to carry out the `mergeAndCountSplitInv` process. It is still $\Theta(n)$, thus our inversion counting algorithm is guaranteed to be $\Theta(n \log n)$.

2.2 Matrix Multiplication

Matrix multiplication is an important mathematical problem.

Input Two matrices X, Y of dimension $N \times N$.

Output Product matrix $Z = X \cdot Y$.

Algorithm 2.2 Merge and Count Split Inversion

input:
A = 1st sorted sub-array, of length $\lfloor n/2 \rfloor$
B = 2nd sorted sub-array, of length $\lceil n/2 \rceil$
output:
C = sorted array of length n
numSplitInv = number of split inversions

```
1:  i = 1, j = 1, numSplitInv = 0
2:  for k = 1 to n do
3:      if i > A.len then                          ▷ A has been exhausted
4:          C[k] = B[j++]
5:      else if j > B.len then                     ▷ B has been exhausted
6:          C[k] = A[i++]
7:      else if A[i] ≤ B[j] then   ▷ In this case equality is actually impossible.
8:          C[k] = A[i++]
9:      else                                       ▷ A[i]>B[j]
10:         C[k] = B[j++]
11:         numSplitInv += A.len
```

The definition of matrix multiplication is $Z_{ij} = \sum_{k=1}^{N} X_{ik} Y_{kj}$. Calculating the product matrix directly will result in a $\Theta(n^3)$ algorithm.[1] We will introduce an ingenious divide-and-conquer algorithm developed by Strassen that is more efficient.

At first sight, it might seem plausible to divide each matrix into 4 sub-matrices of dimension $N/2 \times N/2$ in the divide phase of the divide-and conquer process:

$$XY = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}.$$

However, this division does not make great difference. The algorithm is still $\Theta(n^3)$, as we will prove later. Recall that in the Karatsuba Multiplication algorithm, we reduced the number of products by 1 by applying the Gauss's trick: obtain some products by linear combination of other products, rather than direct multiplication. Since addition/subtraction is generally more efficient than multiplication, it usually pays off to appropriately choose the products to calculate in order to reduce the number of products to calculate. In the naive divide-and-conquer design above, we have to calculate 8 products of sub-matrices. Strassen's brilliant algorithm reduces this number to 7, as shown in Algorithm 2.3, and ends up with smaller time consumption. The explanation of its time complexity, as well as that of the naive divide-and-conquer algorithm will be addressed later.

[1] Here the input size is $\Theta(n^2)$, rather than $\Theta(n)$.

Algorithm 2.3 Strassen's Matrix Multiplication

input:
Two matrices X, Y of dimension $N \times N$

output:
Matrix product $X \cdot Y$

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

2: Recursively calculate

$$P_1 = A(F - H),\ P_2 = (A + B)H,\ P_3 = (C + D)E,\ P_4 = D(G - E)$$
$$P_5 = (A + D)(E + H),\ P_6 = (B - D)(G + H),\ P_7 = (A - C)(E + F)$$

3: Linearly combine the products in step 2 to obtain XY:

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

2.3 Closest Pair

The closest pair problem is the first computational geometry problem we meet.

Input A set of n points $P = \{p_1, \ldots, p_n\}$ on the $\mathbb{R}^2$ plane. For simplicity, we assume that they have distinct x coordinates and y coordinates.

Output A pair of distinct points $p^*, q^* \in P$ that minimizes the Euclidean distance between two points $d(p, q)$ with $p, q \in P$.

A brute-force algorithm is obviously $\Theta(n^2)$. A divide-and-conquer approach can improve it to $\Theta(n \log n)$. Its subtlety lies, as usual, in the 3rd step: combination of solutions to the sub-problems. As preparation before the divide-and-conquer, we sort the points respectively by x and y coordinates, and note the results as P_x, P_y. The sort process is $\Theta(n \log n)$ using merge sort, thus we can obtain a $\Theta(n \log n)$ algorithm as long as the divide-and-conquer process takes no more than $\Theta(n \log n)$.

The skeleton of the process is shown in Algorithm 2.4. `ClosestSplitPair` remains to be illustrated. It outputs the "split pair", i.e. one point in Q and the other in R, with minimum distance.

Let $\overline{x}$ represent the largest x coordinate in Q, i.e. in the left half of P. Since we have P_x, $\overline{x}$ can be obtained in $O(1)$ time. Define S_y as points in P with x coordinate inside $[\overline{x} - \delta, \overline{x} + \delta]$, sorted by y coordinate. We have the following lemma.

Lemma 4. *Let $p \in Q, q \in R$ be the split pair with $d(p, q) < \delta$. Then we must have*

Algorithm 2.4 Closest Pair Searching ClosetPair(P_x, P_y)

input:

A set of n points $P = \{p_1, \ldots, p_n\}$ on $\mathbb{R}^2$, sorted respectively by x and y coordinates as P_x and P_y

output:

p, q with minimum Euclidean distance

1: Let Q be the left half of P and R be right half of P. According to P_x, P_y, form Q_x, Q_y, R_x, R_y, i.e. Q, R sorted by x and y coordinates. ▷ $\Theta(n)$
2: $(p_1, q_1) = \text{ClosestPair}(Q_x, Q_y)$
3: $(p_2, q_2) = \text{ClosestPair}(R_x, R_y)$
4: $\delta = \min\{d(p_1, q_1), d(p_2, q_2)\}$
5: $(p_3, q_3) = \text{ClosetSplitPair}(P_x, P_y, \delta)$ ▷ Should be $O(n)$
6: Return the best among $(p_1, q_1), (p_2, q_2)$ and (p_3, q_3)

- $p, q \in S_y$;
- *p, q are at most 7 positions away from each other in S_y.*

Proof. Let $p(x_1, y_1) \in Q$, $q(x_2, y_2) \in R$, and we have

$$d(p, q) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} < \delta.$$

Since $\overline{x}$ is the largest coordinate in Q, we have

$$x_1 \leq \overline{x} \leq x_2.$$

Thus

$$|x_2 - \overline{x}| = x_2 - \overline{x} \leq x_2 - x_1 \leq \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} < \delta,$$
$$|x_1 - \overline{x}| = \overline{x} - x_1 \leq x_2 - x_1 \leq \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} < \delta.$$

Which leads to the conclusion

$$x_1, x_2 \in [\overline{x} - \delta, \overline{x} + \delta].$$

This can be directly translated to the first claim of the lemma: $p, q \in S_y$. Figure 2.1 helps to prove the 2nd claim.

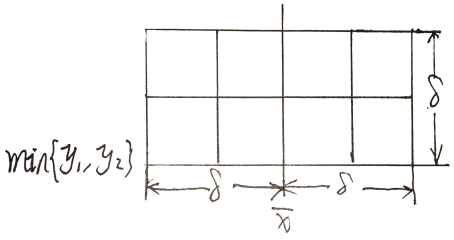


Figure 2.1: p, q at most 7 positions away from each other

In Figure 2.1, we draw 8 $\delta/2 \times \delta/2$ grids around $\overline{x}$ and $\min\{y_1, y_2\}$. Since $|y_1 - y_2| < d(p,q) < \delta$ and $p, q \in S_y$, we know that p, q must be contained in these grids. On the other hand, each grid can contain at most 1 point, because if a grid contained 2 points, their distance would be smaller than $\sqrt{2}/2\delta$, thus smaller than δ, violating the prerequisite that δ is the minimum distance between a non-split pair. As a result, there can be at most 8 points inside these grids, including p and q. Hence p, q are at most 7 points away from each other. □

According to Lemma 4, we finally have the `ClosestSplitPair` algorithm as shown in Algorithm 2.5.

Algorithm 2.5 ClosestSplitPair(P_x, P_y, δ)

input:
P_x, P_y, δ as defined in Algorithm 2.4.
output:
Split pair $p \in Q, q \in R$ with $d(p,q) < \delta$, or *null* if such pair does not exist.

```
1: Initialize best = δ, best_pair = null
2: for i = 1 to |S_y| − 1 do
3:     for j = 1 to min{7, |S_y| − i} do
4:         Let p, q be i^th, (i + j)^th points of S_y
5:         if d(p, q) < best then
6:             best_pair = p, q, best = d(p, q)
```

2.4 The Master Method

Potentially useful algorithmic ideas often need mathematical analysis to evaluate. The master method is a general recurrence approach to analyze the running time of divide-and-conquer algorithms.

2.4.1 Examples

Recall the integer multiplication problem. Let $T(n)$ represent the maximum number of primitive operations needed to multiply two n-digit integers. The recurrence approach aims at expressing $T(n)$ in terms of running time of recursive calls concerning smaller n. In this specific divide-and-conquer algorithm, we hope to express $T(n)$ as function of $T(n/2)$. A recurrence approach needs a base case. In this problem the base case is trivial: $T(1) \leq a$, in which a is a constant.

For the divide-and-conquer algorithm without Gauss's trick, we have

$$T(n) \leq 4 \cdot T(n/2) + O(n). \tag{2.1}$$

When Gauss's trick is applied, we have

$$T(n) = 3 \cdot T(n/2) + O(n). \tag{2.2}$$

Merge sort takes $O(n \log n)$ running time. It has the recurrence relation

$$T(n) = 2 \cdot T(n/2) + O(n). \tag{2.3}$$

We have no idea how to obtain the running time from these recurrence relations, but it is clear that the rank of the three algorithms in terms of running time is (2.1)>(2.2)>(2.3).

2.4.2 Mathematical Statement

The master method helps to obtain the running time of an algorithm according to its recurrence relation. It assumes that all sub-problems have equal size, thus it's not applicable to the closest pair problem, in which the left and right halves of the points are not guaranteed to have the same number of points. We also assume that the base case is trivial: $T(n) \leq a$ (a is constant) for all sufficiently small n. Consider an algorithm that has the recurrence relation

$$T(n) \leq a \cdot T(n/b) + O(n^d),$$

in which a, b, n are constants with clear meanings. a is the number of recursive calls, b is the input size shrinkage factor, and $O(n^d)$ describes the amount of work needed to combine the solutions to the sub-problems. a, b are both larger than 1, while d can be as small as 0. Master method provides the form of $T(n)$ in different cases, as expressed in Theorem 5.

Theorem 5. *$T(n)$ can be expressed in big-O notation as follow[2]:*

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

Now let's look at a few examples.

For merge sort algorithm 1.2 with recurrence relation (2.3), we have $a = 2$, $b = 2$, $d = 1$, thus it belongs to case 1 and has running time $O(n \log n)$.

Consider binary search, which has the recurrence relation

$$T(n) = T(n/2) + O(1),$$

i.e. $a = 1$, $b = 2$, $d = 0$, hence it is $O(\log n)$.

For the integer multiplication algorithm without Gauss's trick (2.1), we have $a = 4$, $b = 2$, $d = 1$, ending up with case 3. Thus it has time complexity $O(n^2)$, i.e. the divide-and-conquer approach fails to improve the time consumption in comparison with the primary school method.

When we take Gauss's trick into account, i.e. with Karatsuba multiplication algorithm 1.1, in (2.2) we have $a = 3$, $b = 2$, $d = 1$. We are still in case 3 and end up with $O(n^{\log_2 3})$, which is smaller than $O(n^2)$ but larger than $O(n \log n)$.

[2]If the recurrence relation is written with = rather than $\leq$, the result will be in Θ notation.

Strassen's algorithm 2.3 for matrix multiplication has the recurrence relation

$$T(n) = 7 \cdot T(n/2) + O(n^2),$$

which leaves us in case 3. Its running time is therefore $O(n^{\log_2 7})$, which is better than $O(n^3)$.

As an illustration of case 2, consider the recurrence relation

$$T(n) \leq 2 \cdot T(n/2) + O(n^2).$$

With $a = b = d = 2$, we are in case 2, and end up with running time $O(n^2)$. In this case, the running time is governed by the work outside the recursive call, i.e. the time spent on combining solutions to the sub-problems dominates the global time consumption.

2.4.3 Proof

We will prove the correctness of the master method in this section. As having been stated above, we assume that the recurrence relation takes the form

- $T(1) \leq c$
- $T(n) \leq a \cdot T(n/b) + cn^d$

It's fine to use the same constant c in both the base case and the recurrence relation because we are using $\leq$. In order to make the process less tedious, we also assume that n is a power of b. The argument will be similar to what we did to obtain the running time of merge sort: through analysis on the recursive tree. Note that in this section when we refer to a value of time consumption, we always mean that the actual time consumption is smaller than or equal to this value.

The recursive tree has $\log_b n + 1$ levels, from level 0 (the original problem) to level $\log_b n$ (trivial problem of size 1). At level j, there are in total a^j sub-problems, each of size n/b^j. For $j \neq \log_b n$, the time consumption at this level is contributed by the cn^d term:

$$a^j \cdot c \cdot \left(\frac{n}{b^j}\right)^d = c \cdot n^d \cdot \left(\frac{a}{b^d}\right)^j.$$

Summing it up over all levels leads to the result $c \cdot n^d \cdot \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^d}\right)^j$. At level $\log_b n$, the time consumption is simply the combination of all base cases:

$$c \cdot a^{\log_b n}$$

The total time consumption is thus

$$c \cdot n^d \cdot \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^d}\right)^j + c \cdot a^{\log_b n} = c \cdot n^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j \tag{2.4}$$

This leads to classified discussion over the value of $\frac{a}{b^d}$.

1. $a = b^d$. In this case, (2.4) becomes

$$c \cdot n^d(\log_b n + 1) = O(n^d \log_b n)$$

2. $a < b^d$. In this case, (2.4) becomes

$$c \cdot n^d \frac{1 - \left(\frac{a}{b^d}\right)^{\log_b n + 1}}{1 - \frac{a}{b^d}} < \frac{c}{1 - \frac{a}{b^d}} n^d = O(n^d)$$

3. $a > b^d$. In this case, (2.4) becomes

$$c \cdot n^d \frac{\left(\frac{a}{b^d}\right)^{\log_b n + 1} - 1}{\frac{a}{b^d} - 1} < \frac{c}{\frac{a}{b^d} - 1} n^d \frac{a}{b^d} \left(\frac{a}{b^d}\right)^{\log_b n} = O(a^{\log_b n}) = O(n^{\log_b a})$$

Therefore we have completed the proof of the master method. □

The essential role that a/b^d plays here comes naturally from the meaning of a, b, d. Each problem produces a sub-problems in the next level. We call a the **rate of sub-problem proliferation, abbr. RSP**. Size of each sub-problem shrinks by b times after each recurrence, and the work load shrinks by b^d times, so we call b^d the **rate of work shrinkage, abbr. RWS**. The three cases of the master method can be interpreted as follow.

1. RSP = RWS. The amount of work at each level is cn^d. With totally $\log_b n$ levels, the problem should be $O(n^d \log_b n)$.

2. RSP > RWS. The amount of work increases with the recursion level j. The last level dominates the running time, thus the overall running time is proportional to the number of sub-problems (base cases) in the last level. The problem is $O(a^{\log_b n})$.

3. RSP < RWS. The amount of work decreases with the recursion level j. The root level dominates the running time, thus the problem is $O(n^d)$.

Chapter 3

Randomized Algorithms

3.1 Quick Sort

3.1.1 Overview

Quick sort is a prevalent sorting algorithm in practice. It is $O(n \log n)$ on average, and it works in place, i.e. extra memory need to carry out the sort is minimal, whereas for merge sort, we need at least $O(n)$ extra memory. The problem is the same as specified for merge sort. Here we assume that all items inside the array are distinct for simplicity.

The key idea of merge sort is **partition the array around a pivot element**. Plenty of deliberation remains for the choice of the pivot element. For the moment we just assume that the first element is used. In a partition, the array is rearranged so that elements smaller than the pivot are put before the pivot, while elements larger than it are put after the pivot. The partition puts the pivot in the correct position. By recursively partition the two sub-arrays on both sides of the pivot, the whole array becomes sorted. As will be revealed later, a partition can be finished with $O(n)$ time and no extra memory. The skeleton of the algorithm is shown in Algorithm 3.1.

3.1.2 Partition Subroutine

If the in place requirement is thrown away, it is easy to come up with a partition algorithm using $O(n)$ time and $O(n)$ extra memory, as shown in Algorithm 3.2.

Now we try to implement an in-place partition algorithm that uses no extra memory. During the process, the array will be composed of 4 consecutive parts: the pivot p at the first position, then elements smaller than p, elements larger than p and finally elements remaining to be partitioned. Algorithm 3.3 provides such an implementation. It completes the partition in one scan of the array, and uses no extra memory. i is the index of the first element larger than p[1],

[1]If $A[2]$ is smaller than p, i will not have the same meaning when it gets initialized to 2.

Algorithm 3.1 Skeleton of Quick Sort

input:
Array A with n distinct elements in any order
output:
Array A in sorted order

```
1: if A.len == 1 then
2:     return
3: else
4:     p = ChoosePivot(A)
5:     Partition A around p
6:     Recursively sort 1st part(on left of p)
7:     Recursively sort 2nd part(on right of p)
```

Algorithm 3.2 Partition with $O(n)$ Extra Memory

input:
Array A with $n(n > 1)$ distinct elements in any order
Pivot element p, put at first position of A
output:
Array A partitioned around p

```
1: Allocate temp[n]
2: small = 1, big = n
3: for i = 2 to n do
4:     if A[i] > p then
5:         temp[big − −] = A[i]
6:     else                                   ▷ A[i] < p
7:         temp[small + +] = A[i]
8: Assert small == big
9: temp[small] = p
10: Copy temp back to A
```

while j is the index of the first unpartitioned element.

3.1.3 Choice of Pivot

The running time of quick sort depends on the choice of the pivot. The naive approach taken above to always choose the first element is not satisfactory. If the array is already sorted, each time the size of the problem is only reduced by 1, which is no better than selection sort, and the running time if $O(n^2)$. This is indeed the worst case of quick sort. On the contrary, if each time the pivot divides the array into sub-arrays of the same size, we will have the recurrence

But the algorithm is still correct. In such case, i remains the same as j until we encounter the first element larger than p, and *swap* will not do anything because k and i are always the same when we do the comparison.

Algorithm 3.3 Partition with No Extra Memory

input:
Array A with $n(n > 1)$ distinct elements in any order
Pivot element p, put at first position of A
output:
Array A partitioned around p

```
1: i = 2, j = 2
2: for k = 2 to n do
3:     if A[k] < p then
4:         swap(A[k], A[i++])
5:     j++
6: swap(A[1], A[i − 1])
```

relation

$$T(n) = 2 \cdot T(n/2) + O(n),$$

and the running time will be $O(n \log n)$ according to the master method, which is the best case.

3.1.4 Quick Sort Theorem

A good solution is to choose the pivot randomly at every recursive call. We will prove the quick sort theorem 6.

Theorem 6. *For every input array A of length n, the average running time of quick sort (with random pivots) is $O(n \log n)$.*

The proof of the theorem involves some basic probability knowledge that won't be covered here. Let Ω represent the sample space of all possible sequences of pivots in quick sort. For any $\sigma \in \Omega$, let $C(\sigma)$ represent the number of comparisons between array elements made by quick sort. We have Lemma 7.

Lemma 7. *The running time of quick sort is dominated by comparisons, i.e. $\exists$ constant c such that $\forall \sigma \in \Omega$,*

$$RT(\sigma) \leq c \cdot C(\sigma).$$

Lemma 7 means that in order to prove the quick sort algorithm, all we need is to prove the expectation of $C(\sigma)$ is $O(n \log n)$. We cannot apply the master method here because in quick sort, the two sub-problems are unlikely to have the same size.

For a fixed input array A, let z_i represent its i^{th} smallest element. Let $x_{ij}(\sigma)$ represent the number of comparisons between z_i and z_j made during quick sort with pivot sequence σ. Whenever a comparison happens, one of the two elements being compared is the pivot. If z_i and z_j have been partitioned respectively into two sides of a pivot before neither is used as pivot, they will never be compared

in the future. If z_i is used as pivot and is compared against z_j, z_i will be at its correct position at the end of the partition, and is excluded from any further comparisons. Thus $\forall i, j, \sigma$, it is clear that $x_{ij}(\sigma)$ is either 0 or 1. A random variable that can only take values 0 and 1, like x_{ij} here, is called an **indicator**.

$C(\sigma)$ can be expressed as the sum over all x_{ij}:

$$C(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} x_{ij}(\sigma).$$

According to the **linearity of expectation**, an taking into account of the fact that x_{ij} is an indicator, we have

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[x_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P[x_{ij} = 1].$$

Here we are actually applying a **decomposition approach** that goes for the analysis of average running time of a lot of randomized algorithms.

1. Identify random variable Y that we care about.
2. Express Y as the sum of a series of indicators: $Y = \sum_l x_l$.
3. Apply linearity of expectation: $E[Y] = \sum_l E[x_l] = \sum_l P[x_l = 1]$.

It can be proved that for any $i < j$,

$$P(x_{ij} = 1) = \frac{2}{j - i + 1}$$

Proof. Consider the set of elements

$$S_{ij} = \{z_i, z_{i+1}, \ldots, z_{j-1}, z_j\}.$$

As long as none of them is chosen as pivot, they will be passed to the same recursive call. If z_i or z_j is the first among them to be chosen as pivot, x_i and x_j will be compared. On the contrary, if any other element is chosen as pivot before z_i and z_j, z_i and z_j will end up in different sub-arrays, and they can never be compared in the future. So $P(x_{ij} = 1)$ is equal to the probability that z_i or z_j is the first element of S_{ij} to be chosen as pivot, i.e. $\frac{2}{j-i+1}$. □

Then we have

$$\begin{aligned} E[C] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P[x_{ij} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\ &= 2 \cdot \sum_{i=1}^{n-1} \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n - i + 1} \right) \end{aligned}$$

$$
\begin{aligned}
&= 2 \cdot \left(\frac{n}{2} + \frac{n-1}{3} + \frac{n-2}{4} + \cdots + \frac{1}{n} \right) \\
&\leq 2n \sum_{k=2}^{n} \frac{1}{k} \\
&\leq 2n \int_{1}^{n} \frac{1}{x} dx = 2n \ln n
\end{aligned}
$$

Thus the running time of quick sort with random pivots is $O(n \log n)$.

3.2 Randomized Selection

3.2.1 RSelect Algorithm

In this section we will discuss the selection problem.

Input Array A containing n distinct numbers in arbitrary order, and number $i \in \{1, 2, \ldots, n\}$.

Output The i^{th} order statistic, i.e. i^{th} smallest element of A.

Obviously, the i^{th} order statistic can be found by sorting the array and then directly pick the i^{th} element. The overall running time is at least $O(n \log n)$. A randomized approach similar to that of quick sort can reduce the average running time to $O(n)$, which is quite amazing because simply reading all elements of the array is already an $O(n)$ process. The algorithm is provided in Algorithm 3.4.

Algorithm 3.4 Randomized Selection

input:
Array A with n distinct elements in any order
Integer $i \in \{1, 2, \ldots, n\}$
output:
i^{th} order statistic of A

```
1:  function RSelect(A, i)
2:      if A.len == 1 then
3:          return A[1]
4:      Randomly choose pivot p from A
5:      Partition A around p
6:      Let j = new index of p, A_1, A_2 = 1st and 2nd part of A
7:      if j == i then
8:          return p
9:      else if j > i then
10:         return RSelect(A_1, i)                    ▷ A_1.len = j − 1
11:     else
12:         return RSelect(A_2, i − j)                ▷ j < i. A_2.len = n − j
```

3.2.2 Running Time

RSelect is also based on recursively partitioning the arrays, but at most one recursive call is needed in each iteration because the i^{th} order statistic can only be on one side of the pivot. As with quick sort, the worse case happens when each time the smallest element is chosen as pivot. The worst running time is $O(n^2)$. The average running time is given by Theorem 8.

Theorem 8. *For any input array A of length n, the average running time of RSelect is $O(n)$.*

In *RSelect*, the only workload outside of the recursive call is the partition, which is an $O(n)$ process. Another way to note its time complexity is that it uses fewer than cn operations, in which c is a constant.

Definition 5. *RSelect is said to be in phase j if size of the current array is between $(3/4)^{j+1}n$ and $(3/4)^j n$.*

Let x_j represent the number of recursive calls during phase j. Then we have

$$RT(RSelect) \leq cn \sum_j x_j \cdot \left(\frac{3}{4}\right)^j \tag{3.1}$$

Now let's consider $E(x_j)$. If *RSelect* chooses a pivot giving a 25%-75% or better partition, the execution is guaranteed to enter the next phase, because the length of the sub-array to examine will shrink by at least 25%. The probability for such a partition to happen is obviously 50%. The problem is analogous to the coin flipping problem. The expectation of x_j is the same as the expectation of the number of flips needed to have the "head" side of the coin show up for the first time. The coin flipping problem is a geometric distribution problem, in which the expectation is $\frac{1}{p} = \frac{1}{1/2} = 2$. Thus $E(x_j) = 2$. Substitute this into (3.1), we have

$$E(RT(RSelect)) \leq cn \sum_j E(x_j) \left(\frac{3}{4}\right)^j = 2cn \sum_j \left(\frac{3}{4}\right)^j \leq 8cn.$$

3.3 Deterministic Selection

In this section we will cover a deterministic selection algorithm that is also $O(n)$. It is not a good choice when compared against random selection because the constant for the big-O notation is larger, and it is not in place. Yet it is still a brilliant and interesting algorithm.

When partitioning an array, the ideal pivot is its median. In order to guarantee the efficiency of the algorithm, we need to find a good enough pivot. The key idea is to use the "median of medians". The array is broken into $n/5$ sub-arrays of length 5. The sub-arrays are sorted, their medians are put into a new array C, and we recursively compute the median of C. The final result is returned as

Algorithm 3.5 Deterministic Selection

input:
Array A with n distinct elements in any order
Integer $i \in \{1, 2, \ldots, n\}$
output:
i^{th} order statistic of A

1: **function** $DSelect(A, i)$
2: $n = A.len$
3: **if** $n \leq 5$ **then**
4: sort A and pick the i^{th} element
5: Break A into $n/5$ sub-arrays of length 5 and sort each sub-array ▷ $O(n)$
6: Let array C store the medians of the $n/5$ sub-arrays ▷ $O(n)$
7: $p = DSelect(C, n/10)$ ▷ Recursively compute median. $T(n/5)$
8: Partition A around p ▷ $O(n)$
9: Let j = new index of p, A_1, A_2 = 1st and 2nd part of A
10: **if** $j == i$ **then**
11: return p
12: **else if** $j > i$ **then**
13: return $DSelect(A_1, i)$ ▷ $T(?)$ running time to be determined
14: **else**
15: return $DSelect(A_2, i - j)$

the pivot. The algorithm is shown in Algorithm 3.5. Two recursive calls are made in Algorithm 3.5. Let $T(n)$ represent the running time of $DSelect$ on an array of length n, then there exists constant c such that

- $T(1) = 1$
- $T(n) \leq cn + T(n/5) + T(?)$

in which the running time of the second recursive call needs to be determined.

Lemma 9. *The 2nd recursive call (line 13 or 15) is guaranteed to be* $O(\frac{7}{10}n)$.

Proof. Let $k = n/5$, and let x_i represent the i^{th} smallest of the k medians. Then the final pivot is $x_{k/2}$. Our goal is to prove that at least 30% of the elements in the input array are smaller than the $x_{k/2}$, and at least 30% are bigger than it. Let's put all elements of the array in a 2D grid as follow, with each sorted group as a column, and x_i in ascending order.

$$\begin{array}{ccccccc}
\circ & \circ & \cdots & \circ & \cdots & \circ & \circ \\
\circ & \circ & \cdots & \circ & \cdots & \circ & \circ \\
x_1 & x_2 & \cdots & x_{k/2} & \cdots & x_{k-1} & x_k \\
\circ & \circ & \cdots & \circ & \cdots & \circ & \circ \\
\circ & \circ & \cdots & \circ & \cdots & \circ & \circ
\end{array}$$

Obviously, elements in red color are smaller than $x_{k/2}$, while elements in blue color are bigger than it. Both include $\frac{3}{5} \times \frac{1}{2} = 30\%$ of the elements. Thus the size of the sub-array is guaranteed to shrink by at least 30% in the 2nd recursive call, so its running time is $O(\frac{7}{10}n)$. □

Theorem 10. $\forall$ *input array of size* n*, DSelect runs in* $O(n)$ *time.*

Proof. Now we have

$$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right).$$

The master method is not applicable because the two sub-problems are not of the same size. We will prove $T(n) \leq 10cn$ by induction.

The base case($n = 1$) is trivial. Suppose that $T(k) \leq 10ck$ holds for all $k < n$. Then we have

$$\begin{aligned} T(n) \leq & cn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) \\ \leq & cn + 10c \cdot \frac{n}{5} + 10c \cdot \frac{7n}{10} \\ = & 10cn \end{aligned}$$

Hence $T(n)$ is $O(n)$. □

3.4 Lower Bound for Comparison-Based Sorting

Up to now we have in our toolbox $O(n \log n)$ sorting algorithms and $O(n)$ selection algorithms. In this section we will prove that we cannot do better with comparison-based sorting.

Definition 6. *In a comparison-based sorting, the array elements can only be accessed via comparisons.*

Merge sort, heap sort, quick sort, selection sort are all examples of comparison-based sorting. Bucket sort, counting sort, radix sort are examples of non-comparison-based sorting. They require extra knowledge of the data to sort.

The following theorem provides a lower bound for the running time of comparison-based sorting.

Theorem 11. *Every comparison-based sorting algorithm has worst-case running time* $\Omega(n \log n)$*.*

Proof. Consider the action of a comparison-based sorting algorithm on an input composed of integers from 1 to n. There are totally $n!$ such inputs. Let k be the minimum number of comparisons needed to address all these inputs. Each comparison has two possible results, thus k comparisons have 2^k possible

combinations of results. In order for all the $n!$ possible inputs to be sorted correctly, we must have

$$2^k \leq n!,$$

because otherwise there would be at least 2 inputs that appeared the same for the comparison-based algorithm, according to the pigeonhole principle[2]. Hence we have

$$k \leq \log(n!) < \log(n/2)^{n/2} = \frac{n}{2} \log \frac{n}{2},$$

which means that the running time of the algorithm is $\Omega(n \log n)$.

□

[2] Also named drawer principle

Chapter 4

Graph Primitives

Graphs represent pairwise relationships amongst a set of objects. The objects are called vertices or nodes. The relationships are called edges or arcs, each connecting a pair of vertices. An edge can be directed or undirected. The set of vertices and the set of edges are noted respectively as V and E. Graph is a concept heavily used in reality. Road networks, the web, social networks, precedence constraints are all examples of graphs.

A connected graph composed of n vertices with no parallel edges has at least $n-1$ and at most $n(n-1)/2$ edges. Let m represent the number of edges. In most applications, m is $\Omega(n)$ and $O(n^2)$. If m is $O(n)$ or close to it, the graph is called a sparse graph, while if m is closer to $O(n^2)$, it's called a dense graph. Yet their delimitation is not strictly clear.

4.1 Representation

4.1.1 Adjacent Matrix

An undirected graph G with n vertices and no parallel edges can be represented by an $n \times n$ 0-1 matrix A. $A_{ij} = 1$ when and only when G has an $i - j$ edge. Variants of this representation can easily accommodate parallel edges, weighted edges: just let A_{ij} represent the number of parallel edges or the weight of the edge. For directed graphs, $i \to j$ can be represented by $A_{ij} = 1$ and $A_{ji} = -1$.

Adjacent matrix representation requires $\Theta(n^2)$ space. For a dense graph this is fine, but for a sparse graph it is wasteful.

4.1.2 Adjacent Lists

The adjacent lists representation is composed of 4 ingredients:

- Array/List of vertices. $\Theta(n)$ space.
- Array/List of edges. $\Theta(m)$ space.

- Each edge points to its end points. $\Theta(m)$ space.
- Each vertex points to edges incident on it. $\Theta(m)$ space.

Adjacent lists representation requires $\Theta(n+m)$ space.

The choice between the two representations depends on the density of the graph and operations to take. We will mainly use adjacent lists in this chapter.

4.2 Minimum Cut

4.2.1 Definition

Definition 7. *A cut of a graph (V, E) is a partition of V into two non-empty sets A and B.*

A graph with n vertices has $2^n - 2$ possible cuts.

Definition 8. *The crossing edges of a cut(A, B) are those with*

- *one endpoint in A and the other in B, for undirected graphs;*
- *tail in A and head in B, for directed graphs.*

We will try to solve the minimum cut problem:

Input An undirected graph $G = (V, E)$ in which parallel edges are allowed.

Output A cut (A, B) with minimum number of crossing edges.

A lot of problems in reality can be reduced to a minimum cut problems:

- Identify weakness point of physical networks;
- Community detection in social networks;
- Image segmentation.

4.2.2 Random Contraction Algorithm

Algorithm 4.1 provides a random approach to find a cut.

4.2.3 Probability of Correctness

Algorithm 4.1 is not guaranteed to always return a minimum cut. We have to iterate the whole algorithm multiple times and choose the cut with minimum number of crossing edges obtained during the process. In order to estimate the number of iterations needed, we will calculate the probability that a specific minimum cut (A, B) is returned in one iteration.

If one of the crossing edges of (A, B) is selected in step 2, the minimum cut cannot be returned. If there are k crossing edges in (A, B), the probability that

Algorithm 4.1 Random Contraction

input:
An undirected graph $G = (V, E)$ in which parallel edges are allowed.
output:
A cut (A, B) with minimum number of crossing edges
1: **while** There are more than 2 vertices left **do**
2: Pick a remaining edge (u, v) randomly
3: Merge(or contract) u, v into a single vertex
4: Remove self-loops
5: Return cut represented by 2 final vertices

a crossing edge is selected at the first iteration is k/m. The number of edges decrease by an indefinite number at each iteration, while the number of vertices decrease by exactly 1 each time. Thus it is preferable to express the probability in terms of n instead of m.

Each vertex v is related to a cut $(v, V - v)$. The number of crossing edges of this cut is the number of edges incident on v, i.e. the degree of v. Considering the definition of minimum cut, this number must be larger than k. We also know the sum of the degrees of all vertices: $\sum_v degree(v) = 2m$. Thus we have $2m = \sum_v degree(v) \geq kn$. Hence the probability that a crossing edge is not selected at the first iteration in step 2 is

$$1 - k/m \geq 2/n.$$

At the second iteration, the same argument still holds. Let m' represent the number of remaining edges after the first iteration. We have $2m' = \sum_v degree(v) \geq k(n-1)$. Thus the probability that a crossing edge is selected at the 2nd iteration under the condition that no crossing edge was selected at the 1st iteration is

$$1 - k/m' \geq 2/(n-1)$$

.

The same argument continues further until the last iteration, i.e. the $(n-2)^{th}$ iteration. Finally, the probability that no crossing edge is selected in the whole process, thus resulting in the minimum cut (A, B) is

$$\begin{aligned} P(A,B) &\geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right)\cdots\left(1 - \frac{2}{n-(n-3)}\right) \\ &= \frac{(n-2)\cdot(n-3)\cdots 2\cdot 1}{n\cdot(n-1)\cdots 4\cdot 3} \\ &= \frac{2}{n(n-1)} > \frac{1}{n^2}. \end{aligned}$$

The probability seems small, but it already makes great advancement when compared with brute-force method, in which case the probability to obtain (A, B) in an iteration is $\frac{1}{2^n}$.

After N iterations, the probability that (A, B) has not been found is

$$P(not\ found) < \left(1 - \frac{1}{n^2}\right)^N \leq e^{-\frac{N}{n^2}}.$$

If $N = n^2$, the probability is smaller than $1/e$. If $N = n^2 \ln n$, it is smaller than $1/n$.

4.2.4 Number of Minimum Cuts

A graph can have multiple minimum cuts. For example, a tree with n vertices has $n-1$ minimum cuts. We would like to find the largest number of minimum cuts that a graph with n vertices can have.

Theorem 12. *A graph with n vertices can have at most $\binom{n}{2}$ minimum cuts.*

Proof. Consider the graph in which the n vertices form a circle. Removing any two edges results in a minimum cut. Thus in total it has $\binom{n}{2}$ minimum cuts. The rest of the proof aims at proving that a graph with n vertices cannot have more minimum cuts.

Recall that the probability for Algorithm 4.1 to return a specific minimum cut is bigger than $\frac{2}{n(n-1)}$. Suppose there are k minimum cuts. The events "Algorithm 4.1 returns minimum cut C_i" and "Algorithm 4.1 returns minimum cut C_j" are disjoint events when $i \neq j$. Thus we have

$$1 \geq P(\text{return a minimum cut}) \geq \frac{2k}{n(n-1)}.$$

Therefore,

$$k \leq \frac{n(n-1)}{2} = \binom{n}{2}.$$

□

4.3 Breadth First Search

Graph search is widely used for various purposes:

- Check if a network is connected;
- Find shortest paths, e.g. for driving navigation, or formulating a plan;
- Calculate connected components.
- ...

We will introduce a few fast algorithms based on graph search. Graph search usually starts from a source vertex. When searching the graph, we want to find everything that is findable, i.e. every vertex reachable from the source via a path. Moreover, we never explore anything twice. In terms of running time, our goal is $O(m+n)$.

BFS explores the nodes of a graph in layers. Nodes with the same distances from the source are in the same layer. It can be used to compute shortest paths of graphs, and to compute connected components of undirected graphs. It guarantees $O(m+n)$ running time. The general pattern of BFS is shown in Algorithm 4.2.

Algorithm 4.2 Breadth First Search(BFS)

input:
Graph G with all vertices unexplored
Source vertex s
output:
G with all vertices reachable from s explored.

```
1: Mark s as explored.
2: Let Q = queue initialized with s
3: while Q ≠ ∅ do
4:     Remove first element v of Q
5:     for each edge (v, w) do
6:         if w is unexplored then
7:             Mark w as explored
8:             Add w to Q
```

At the end of BFS, the fact that a node v is explored means the existence of a path from s to v.

4.3.1 Shortest Path

Algorithm 4.3 calculates the shortest path from s to any vertex reachable from s.

After the algorithm terminates, $dist(v) = i$ means that v is in the i^{th} layer and that the shortest path connecting s and v has i edges.

4.3.2 Undirected Connectivity

Definition 9. *For an undirected graph $G(V, E)$, connected components are equivalence classes of the equivalence relation*[1] *$u \sim v$, in which u, v are its vertices and $u \sim v \iff \exists$ path from u to v.*

Identifying connected components of graphs is useful for various purposes:

[1] An equivalence relation on a set must satisfy: 1. $a \sim a$; 2. If $a \sim b$, then $b \sim a$; 3. If $a \sim b$ and $b \sim c$, then $a \sim c$.

Algorithm 4.3 Shortest Path - BFS

input:
Graph G with all vertices unexplored
Source vertex s
output:
$dist(v)$ for any vertex v, i.e. min number of edges on a path from s to v

1: Initialize $dist(v) = \begin{cases} 0, & if(v == s) \\ +\infty, & if(v \neq s) \end{cases}$
2: Mark s as explored.
3: Let Q = queue initialized with s
4: **while** $Q \neq \emptyset$ **do**
5: Remove first element v of Q
6: **for** each edge (v, w) **do**
7: **if** w is unexplored **then**
8: Mark w as explored
9: $dist(w) = dist(v) + 1$
10: Add w to Q

- Check if a network is disconnected;
- Graph visualization;
- Clustering.

When it comes to the calculation of connected component, undirected graphs and directed graphs are significantly different. BFS is an effective method for calculating the connectivity of undirected graphs. Algorithm 4.4 computes the CCs of an undirected graph in $O(m + n)$ time.

Algorithm 4.4 Connected Components of Undirected Graph - BFS

input:
Undirected graph G with all vertices unexplored and labeled 1 to n
output:
Connected components of G

1: **for** $i = 1$ **to** n **do**
2: **if** i not explored **then**
3: $BFS(G, i)$ ▷ discovers i's connected component

4.4 Depth First Search

DFS is a more aggressive method to search an graph than BFS. It explores the nodes following the edges as deeply as possible, and only backtracks when necessary. DFS is especially important for dealing with directed graphs. As we

are about to demonstrate, it helps to compute topological ordering of directed acyclic graphs and strongly connected components of directed graphs. As with BFS, DFS also runs in $O(m+n)$ time.

DFS can be implemented by mimicking BFS in Algorithm 4.2. A stack should be used instead of a queue. A recursive approach is shown in Algorithm 4.5. DFS can also be used to calculate connected components of undirected

Algorithm 4.5 Depth First Search (Recursive)

input:
Graph G with all vertices unexplored
Source vertex s
output:
G with all vertices reachable from s explored.

```
1: function DFS(Graph G, node s)
2:     Mark s as explored
3:     for each edge (s, v) do
4:         if v not explored then
5:             DFS(G, v)
```

graphs. But we will focus on two applications of DFS that cannot be handled with BFS.

4.4.1 Topological Sort

Topological sort aims at putting the nodes of a directed graph in topological ordering.

Definition 10. *A topological ordering of a directed graph G is a labeling f of G's nodes among $\{1, 2, \ldots, n\}$ such that $\forall (u, v) \in G,\ f(u) < f(v)$.*

All edges of a directed graph go forward in a topological ordering. Topological sort is applied when the order of a sequence of tasks with precedence constraints needs to be arranged. Note that if a directed graph contains a cycle, there exists no topological ordering for it. This condition is actually necessary and sufficient.

Theorem 13. *A directed graph has a topological ordering if and only if it contains no cycle.*

One way to find a topological ordering of a DAG is by identifying sink nodes, i.e. nodes with no outgoing arcs.

Theorem 14. *A DAG has at least one sink node.*

Proof. The theorem can be proved by contradiction easily. Suppose we have a DAG containing no sink node. Starting from a source node, we can follow one of its outgoing arcs to a new node. The process can be done indefinitely because

every node has outgoing arcs. But it is inevitable that at least one node will be visited multiple times after $n+1$ steps, forming a cycle, which contradicts with the definition of DAG. □

A sink node is a perfect candidate for the last position in the topological ordering: no arc starts from it. Algorithm 4.6 calculates the topological ordering of a DAG by recursively putting a sink node at the end of the ordering.

Algorithm 4.6 Topological Ordering of DAG

input:
Directed acyclic graph G with n nodes
output:
Topological ordering of G

```
1: function TopologicalOrdering(Graph G)
2:     if G is empty then
3:         return
4:     Find a sink node v in G
5:     set f(v) = n
6:     TopologicalOrdering(G − {v})
```

The algorithm using DFS is shown in Algorithm 4.7. A loop over the nodes is added in order to guarantee the correctness when G is not connected.

Algorithm 4.7 Topological Ordering of DAG - DFS

input:
Directed acyclic graph G with n nodes, unexplored
output:
Topological ordering of G

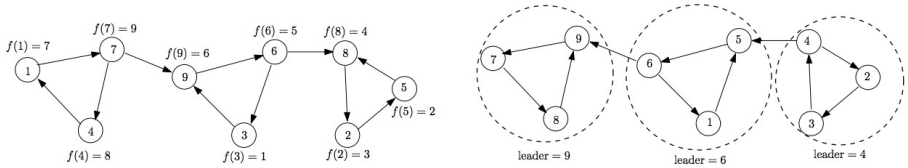
```
1: function DFSLoop(Graph G)
2:     current_label = n
3:     for each v in G do
4:         if v not explored then
5:             DFS(G, v)
6: function DFS(Graph G, node s)
7:     Mark s as explored
8:     for each edge (s, v) do
9:         if v not explored then
10:            DFS(G, v)
11:    f(s) = current_label − −
```

4.4.2 Strongly Connected Components

The concept of connectivity in directed graphs is different from that in undirected graphs.

Definition 11. *Strong connected components (SCCs) of a directed graph $G(V, E)$ are equivalence classes of the equivalence relation $u \sim v$, in which $u, v \in V$ and $u \sim v \iff \exists$ path from u to v as well as from v to u.*

Algorithm 4.8 computes all SCCs of a directed graph by running DFS twice, thus in $O(m + n)$ time. In the end, vertices with the same "leader" are in the same SCC. An example is provided in Figure 4.1.



(a) First DFS-Loop on G^{rev} (b) Second DFS-Loop on G

Figure 4.1: Example of Algorithm 4.8

Algorithm 4.8 Kosaraju's 2-Pass SCC Algorithm - DFS

input:
Directed graph $G(V, E)$ with n nodes labeled 1 to n
output:
SCCs of G

Reverse G to get G^{rev}
1^{st} loop: $DFSLoop(G^{rev})$
2^{nd} loop: $DFSLoop(G)$
function $DFSLoop$(Graph G)
  Set all nodes unexplored
  $t := 0$, $s := null$ ▷ t: finish time. s: current leader
  2^{nd} loop only: Relabel nodes according to $f(i)$
  for $i = n$ **to** 1 **do**
    if i not explored **then**
      2^{nd} loop only: $s := i$
      $DFS(G, i)$
function DFS(Graph G, node i)
  Mark i as explored
  2^{nd} loop only: $leader(i) := s$
  for each arc $i \to j \in E$ **do**
    if j not explored **then**
      $DFS(G, j)$
    1^{st} loop only: $f(i) := ++t$

The SCCs of a directed graph induce a meta-DAG. The meta-nodes are the SCCs, while its arcs are the original arcs between different SCCs, as shown in

Figure 4.2. It is guaranteed to be a DAG because the existence of any loop in the meta-graph will force a few SCCs to collapse into one big SCC.

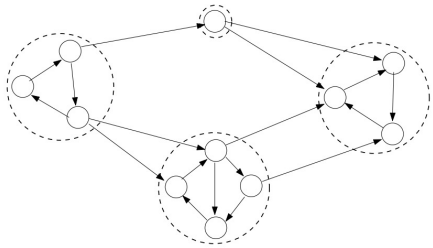


Figure 4.2: Meta-graph induced by SCCs

DFS from a node is guaranteed to reach all other nodes in the same SCC, but it will also reach other SCCs as long as there exist outgoing arcs from this SCC, which are also outgoing arcs in the meta-DAG. Sink nodes in the meta-DAG do not have outgoing arcs, thus the correspondent SCC can be isolated if we start a DFS from one node in this SCC. Algorithm 4.8 does exactly this. In order to prove its correctness we need the following lemma.

Lemma 15. *Consider two adjacent SCCs of G: C_1 and C_2 such that $\exists$ arc $i \to j$ in which $i \in C_1$, $j \in C_2$. Let $f(v)$ denote the finishing time determined in the 1st loop of Algorithm 4.8 for node v. Then we must have*

$$\max_{v \in C_1} f(v) < \max_{v \in C_2} f(v).$$

Proof. In G^{rev}, there exists arc $j \to i$. Let v represent the first vertex in $C_1 \cup C_2$ to be explored. v can be in either C_1 or C_2, as shown in Figure 4.3.

(a) All f-values in C_1 smaller than in C_2 (b) v has the largest f-value in $C_1 \cup C_2$

Figure 4.3: Proof of Lemma 15

If $v \in C_1$, then none of the nodes in C_2 will be explored before all nodes in C_1 are explored, because there exists no arc from C_1 to C_2. Thus we have $f(v_1) < f(v_2)$, $\forall v_1 \in C_1, v_2 \in C_2$, which is a stronger conclusion than our original argument in the lemma.

If $v \in C_2$, DFS for v won't finish before DFSs for all nodes in C_2 finish. In particular, DFS for j won't finish before DFS for all nodes in C_1 finish. Hence

v will have the largest f value amongst all nodes in $C_1 \cup C_2$, and we have $f(v) = \max_{i \in C_2} f(i) > \max_{i \in C_1} f(i)$ □

An obvious corollary of Lemma 15 is as follow.

Corollary 16. $\max_{v \in V} f(v)$ *must lie in a sink SCC, i.e. an SCC that has no outgoing arc.*

Therefore, by starting from the node with the largest f value in the 2^{nd} loop, we are guaranteed to explore a sink SCC of G first. Nodes in this sink SCC are ruled out from further exploration because there have been marked as explored. Every time we set up a new leader, it is guaranteed to be the node with the largest f value amongst all unexplored nodes. DFS from the leader will reach and will only reach nodes in the same SCC as the leader, which is a sink SCC of the unexplored part of the graph. The SCCs will be found one by one.

4.5 Dijkstra's Shortest Path Algorithm

If all edges in a graph have equal lengths, the shortest path problem can be solved by BFS, as discussed in Algorithm 4.3. Dijkstra's algorithm computes shortest paths when edges have different lengths.

Input Directed graph $G(V, E)$. Each edge $e \in E$ has non-negative length l_e. A source vertex s.

Output For each $v \in V$, compute the length of shortest path from s to v in G.

For convenience, we assume that there exists a path from s to any vertex in G.

4.5.1 Algorithm

Dijkstra's algorithm is shown in Algorithm 4.9.

4.5.2 Correctness

The correctness of Dijkstra's algorithm can be proved by induction.

Proof. We will try to prove by induction that after each iteration,$\forall v \in X$, $B[v]$ is the shortest path from s to v, and $A[v]$ is the length of the shortest path.

At the beginning, $X = \{s\}$, $A[s] = 0$, $B[s] =$ *empty path*. Obviously the conclusion is correct. Let's assume that it holds before an iteration, and in this iteration we have chosen the edge $v^* \to w^*$. In order to add w^* to X, we have to prove that $B[v^*] + v^* \to w^*$ with length $A[v^*] + l_{v^*w^*}$ is the shortest path from s to w^*.

Take any path S from s to w^*. It has to cross the boundary between X and $V - X$ somewhere. Suppose the edge from X to $V - X$ is $y \to z$. This path can be divided into 3 segments:

Algorithm 4.9 Dijkstra's Shortest Path Algorithm

input:
Directed graph $G(V, E)$. Each edge $e \in E$ has non-negative lengths l_e
Source vertex s
output:
Shortest path from s to v for all $v \in V$

```
1: Initialize X = {s}                          ▷ X: vertices processed so far
2: A[s] = 0                                    ▷ A[v]: length of shortest path from s to v
3: B[s] = empty path                           ▷ B[v]: shortest path from s to v
4: while X ≠ V do
5:     Among all edges v → w with v ∈ X, w ∉ X, choose v* → w* that
   minimizes A[v] + l_vw                       ▷ Let's call it "Dijkstra's greedy score"
6:     X := X ∪ {w*}
7:     A[w*] := A[v*] + l_{v*w*}
8:     B[w*] := B[v*] + v* → w*
```

1. S_1: from s to y. According to our assumption, it is at least as long as $A[y]$: $L(S_1) \geq A[y]$.
2. S_2: the edge $y \to z$. $L(S_2) = l_{yz}$.
3. S_3: from z to w. All edges have non-negative length, thus $L(S_3) \geq 0$.

Dijkstra's algorithm guarantees that

$$A[v^*] + l_{v^*w^*} \leq A[y] + l_{yz}.$$

Thus we have

$$L(S) = L(S_1) + L(S_2) + L(S_3) \geq A[y] + l_{yz} \geq A[v^*] + l_{v^*w^*}.$$

Therefore, $B[v^*] + v^* \to w^*$ is the shortest path from s to w^*. □

4.5.3 Implementation and Running Time

A naive implementation of Dijkstra's algorithm can take as long as $O(nm)$ time to run: in each iteration, we have to scan through all edges to find $v^* \to w^*$. In order to speed up the execution, we have to turn to the heap data structure.

Heap is a data structure designed to perform insertion and extraction of minimum in $O(\log n)$ time. Conceptually, a heap is an almost perfectly balanced binary tree (null leaves are only allowed at the lowest level). The key of each node must be smaller (or equal to) that of its two children. This property guarantees that the node with the smallest key is at the root. Insertion is performed by adding the element behind the last node and bubbling up, while extraction of minimum is performed by swapping the root and the last node and then bubbling down. The height of the tree is $O(\log n)$, thus insertion and extraction of minimum can be executed in $O(\log n)$ time.

In the implementation of Dijkstra's algorithm, we use a heap to store the vertices in $V - X$. The key of a node is the smallest Dijkstra's greedy score related to this vertex, i.e. for $v \in V - X$, $key[v]$ is the smallest value of $A[u] + l_{uv}, \forall u \in X$. If such edge $u \to v$ does not exist, $key[v] = +\infty$. In each iteration of Dijkstra's algorithm, we extract the minimum of the heap and denote it with w. Now we should have $w \in X$, and $A[w]$ is the length of the shortest path from s to w. Then for all edges $w \to v$ with $v \in V - X$, we update the key of v:

$$key[v] \coloneqq \min\{A[w] + l_{wv}, key[v]\}.$$

If $key[v]$ is changed here, we bubble it down in the heap, which is a $O(\log n)$ operation. In this way the heap gets maintained at each iteration.

In total, we do $n-1$ extractions of minimum, and at most m bubbling down of element in the heap. Each of these operations is $O(\log n)$, thus the total time consumption is $O((m+n)\log n)$. Since the graph is weakly connected ($\forall v \exists$ path from s to v), we have $m \geq n - 1$, hence $O(m + n) = O(m)$. In conclusion, the running time of Dijkstra's algorithm implemented using heap is $O(m \log n)$.

Chapter 5

Data Structures

Data structures help us organize data so that it can be accessed quickly and usefully. Different data structures support different sets of operations, thus are suitable for different tasks.

5.1 Heap

A heap, also named a priority queue, is a container for objects with comparable keys. It should support at least two basic operations: insertion of new object, and extraction(i.e. removal) of the object with minimum[1] key. Both operations are expected to take $O(\log n)$ time. Typical heap implementations also support deletion of an object from the key, which is also $O(\log n)$. The construction of a heap, namely "heapify", takes $O(n)$ rather than $O(n \log n)$.

5.1.1 Use Cases

Heap can be used for sorting. First construct a heap with the n items to be sorted, and then execute extract-min n times. The process takes $O(n \log n)$ time, which is already the optimal running time for comparison based sorting algorithms.

We've already covered the use of a heap to accelerate Dijkstra's algorithm in the previous chapter.

An interesting use case of heap is median maintenance. We define the median of a sorted sequence of n items $x_1, \ldots, x_n$ to be $x_{(n+1)/2}$, for example x_4 for 8 items and x_5 for 9 items.

Input A sequence of unsorted items $x_1, x_2, \ldots, x_n$ provided one-by-one.

Output At each step i, calculate the median of $x_1, \ldots, x_i$ in $O(\log i)$ time.

[1] A heap can also support extraction of object with maximum key, but extract-min and extract-max cannot be supported simultaneously.

The problem can be solved using two heaps, as shown in Algorithm 5.1. For convenience, we assume that the heaps used here supports not only the extraction of min/max, but also checking the key value of the min/max without removing it.

Algorithm 5.1 Median Maintenance using Heaps

input and output:
see above

```
1: Initialize empty MaxHeap that supports extract-max ▷ Stores smaller half
2: Initialize empty MinHeap that supports extract-min ▷ Stores larger half
3: for i = 1 to n do
4:     if x_i < MaxHeap.checkMax() then ▷ Should insert into smaller half
5:         MaxHeap.insert(x_i)
6:     else ▷ insert into larger half
7:         MinHeap.insert(x_i)
8:     if MinHeap.size() - MaxHeap.size() == 2 then ▷ If unbalanced, balance the two halves
9:         MaxHeap.insert(MinHeap.extractMin()
10:    else if MaxHeap.size() - MinHeap.size() == 2 then
11:        MinHeap.insert(MaxHeap.extractMax()
12:    if MinHeap.size() > MaxHeap.size() then ▷ Set median
13:        median = MinHeap.checkMin()
14:    else
15:        median = MaxHeap.checkMax()
```

5.1.2 Implementation

A heap can be conceptually thought of as a binary tree that is as complete as possible, i.e. null leaves are only allowed at the lowest level. The key of any node should be smaller than or equal to keys of its children, if there are any. This guarantees that the object at the root of the tree has the smallest key. This tree can be implemented as an array, with the root at the first position, and nodes at lower levels sequentially concatenated afterwards. If the array A is 1-indexed, then the parent of $A[i]$ is $A[i/2]$, and the children of this node are $A[2i]$ and $A[2i+1]$.

With the array representation of heap, insertion can be implemented as follow:

- Put the new object at the end of the array.
- As long as the key of the new object is smaller than that of its parent, bubble it up.

And extract-min can be implemented as follow:

- Remove the root.

- Move the last object in the array to the first position.
- As long as the key of the object at the root is larger than that of at least one of its children, sink it down. If the keys of both children are smaller, the child with smaller key should be used in the sink-down.

The height of the tree is $O(\log n)$, thus either bubble-up or sink-down can be executed at most $O(\log n)$ times, which guarantees that the two operations take $O(\log n)$ running time.

5.2 Binary Search Tree

Sorted array supports quick search of an element in $O(\log n)$ time, but it takes $O(n)$ time to insert or delete an element. Binary search tree is a data structure that supports both quick search and quick insertion / deletion.

5.2.1 Basic Operations

Each node of a BST contains the key and three pointers to other nodes: the left child, the right child and the parent. Some of the three pointers can be null. The most important property of BST is that for any node, all nodes in its left child has smaller keys than itself, while all nodes in its right key has larger keys. The height of a BST is at least $\log n$ and at most n. A **balanced** BST supports search, insertion and deletion in $O(\log n)$ time. But if it's not balanced, these operations can take as long as $O(n)$ time. Some of its basic operations are listed below.

search In order to search for a node with a specific key value k:

- Start from the root node.
- If a node is null or its key is equal to k, return this node.
- If k is smaller than its key, recursively search its left child.
- If k is larger than its key, recursively search its right child.

insert In order to insert a new node with key value k:

- Start from the root node.
- If the node is null, make a new node here with key value k.
- If k is smaller than its key, go to its left child.
- If k is larger than its key, go to its right child.

max In order to obtain the node with the maximum key value:

- Start from the root node.
- If the node has right child, go to its right child.
- Return the node.

min Similar to max.

successor In order to obtain the successor of a node with key value k:

- If the node has right child, return the max of its right node.
- Otherwise recursively go to its parent, until the key becomes larger than k.

predecessor Similar to successor.

in order traversal In order to traverse all nodes of a BST in order:

- Start from the root node.
- If the node is null, stop.
- Recursively traverse the left child.
- Do something to the node, e.g. print its key.
- Recursively traverse the right child.

delete In order to delete a node with key value k:

- Search for the node.
- If it has no child, change it to null.
- If it has 1 child, replace it with its child.
- If it has 2 children, find its predecessor, who is guaranteed to have at most 1 child, and swap their keys. Then delete the node (currently at its predecessor's old position).

Sometimes a tree node can contain some information about the tree itself, for example the size of the subtree that uses this node as root. For each node n, we have

$$size(n) = size(n.left) + size(n.right) + 1.$$

With this information, we can find the node with the i^{th} largest key among all nodes:

- Start from the root node.
- If $size(n.left) = i - 1$, return the node.
- If $size(n.left) > i - 1$, return the node with the i^{th} largest key in the left subtree.
- If $size(n.left) < i - 1$, return the node with the $(i - size(n.left) - 1)^{th}$ largest key in the right subtree.

5.2.2 Red-Black Tree

The height of a BST can vary between $O(\log n)$ and $O(n)$. Balanced BSTs are guarantees to have $O(\log n)$ height, thus ensuring the efficiency of operations on it. Red-black trees is an implementation of balanced BST. In addition to the key and pointers to the parent and children, nodes in a red-black tree also stores a bit to indicate whether the node is red or black. The following conditions are satisfied by a red-black tree:

1. Each node is either red or black;
2. The root is black;
3. There can never be two red nodes in a row, i.e. red nodes must have black parents and children;
4. Every root $\rightarrow$ null path has the same number of black nodes.

Theorem 17. *The height of a red-black tree with n nodes is smaller than or equal to $O(2\log(n+1))$.*

Proof. Suppose all root $\rightarrow$ null paths contain k black nodes. Then the red-black contains at lest k complete levels, because otherwise there would exist root $\rightarrow$ null paths with fewer than k nodes, thus of cause fewer than k black nodes. Therefore we have

$$n \geq 1 + 2 + \cdots + 2^{k-1} = 2^k - 1,$$

which means $k \leq \log(n+1)$. Suppose the height of the tree is h. According to condition 3, we hereby come to the conclusion

$$h \leq 2k \leq 2\log(n+1).$$

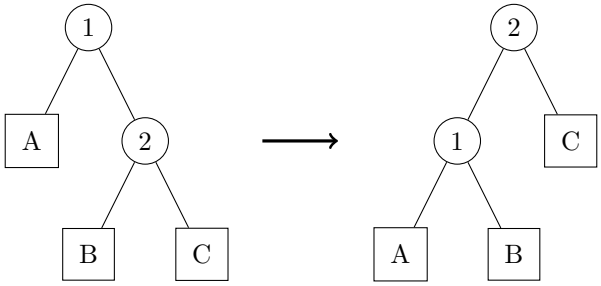
□

An important idea in the implementation of red-black tree is rotation, as illustrated in Figure 5.1. It alters the structure of the tree in a way that makes the tree more balanced, whilst preserves the BST property.

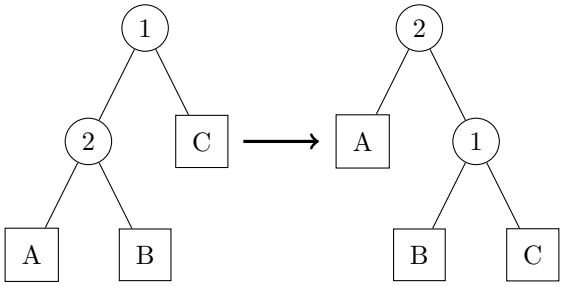
Insertion and deletion in a red-black tree is carried out in two steps. First a normal BST insertion / BST is executed. It is probable that some of the conditions of red-black tree will be violated, thus we then modify the tree by recoloring the nodes and rotations in order to restore the conditions.

When we insert a node into the red-black tree as we do for any BST, we first try to color it as red. If condition 3 is not violated, then everything is fine. Otherwise we wind up in two possible cases, as shown in Figure 5.2, in which x is the newly inserted node.

In case 1, all we need to do is a recoloring of the nodes. The red node is propagated upwards, which may possibly induce another violation of 3. The process can last as much as $O(\log n)$ times until we reach the root. If the root is colored red, condition 2 will be violated, and the solution is to color it back to black.



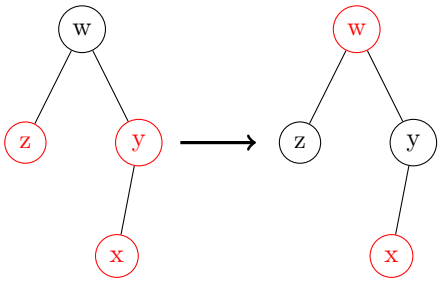
(a) left rotation



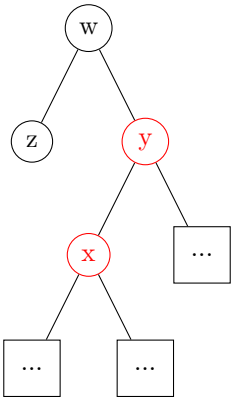
(b) right rotation

Figure 5.1: Rotations in Red Black Tree

During the upward propagation process, it is possible that we meet case 2. Tackling this case is a little bit more complex, but it can be proven that the conditions can be restored via 2-3 rotations and recolorings in $O(1)$ time.



(a) case 1



(b) case 2

Figure 5.2: Insertion in a Red-Black Tree

5.3 Hash Table

5.3.1 Concepts and Applications

Hash table is a data structure designed to efficiently maintain a (possibly evolving) set of items, such as financial transactions, IP addresses, people associated with some data, etc. It supports insertion of a new record, deletion of existing records, and lookup of a particular record (like a dictionary). Assuming that the hash table is properly implemented, and that the data is non-pathological, all these operations can be executed in $O(1)$ time: amazingly fast!

Let's first introduce a few typical use cases of hash table before diving into its implementation.

Hash table can be used to solve the de-duplicates problem.

Input A stream of objects.

Output Unique objects in the stream, i.e. the objects with all duplicates removed.

The problem arises when we want to record the number of unique visitors to a website, or when we want to remove duplicates in the result of a search. With a hash table on the objects implemented, we can solve the problem in linear time. Just examine the objects one by one. For each object x, do a lookup in the hash table H. If x is not found in H, insert it into H and append it to the result, otherwise just continue with the next object.

Another application is the 2-sum problem.

Input An unsorted array A of n integers, and a target sum t.

Output Whether there exists two numbers $x, y \in A$ such that $x + y == t$.

A naive enumerative solution is $O(n^2)$. If we sort A and then search for $t - x$ in A for every $x \in A$, the time consumption can be reduced to $O(n \log n)$. But with hash table, the problem can be solved in merely $O(n)$ time. Just insert all items into the hash table, and then for each $x \in A$ check if $t - x$ is in A via a hash table lookup.

In the early days of compilers, hash table was used to implemented symbol tables. The administrator of a network can use hash table to block certain IP addresses. When exploring huge game trees of chess or Go, hash table can be used to avoid duplicate explorations of the same configuration that can appear enormous times in the tree. In the last case, the size of the tree is so large that hash table is the only plausible method to record whether a configuration has been explored.

5.3.2 Implementation

When implementing hash table, we should think of a generally really big universe U (e.g. all IP addresses, all names, all chessboard configurations, etc), of which we wish to maintain a evolving subset S of reasonable size. The general approach is as follow.

1. Pick n as the number of "buckets". n should be of size comparable with S.
2. Choose a hash function $h : U \to \{0, 1, 2, \ldots, n-1\}$.
3. Use array A of length n to store the items. x should be stored in $A[h(x)]$.

Definition 12. *For a specific hash function h on a universe U, we say there is a collision if $\exists$ distinct $x, y \in U$ such that $h(x) = h(y)$.*

Think of the famous "same birthday" problem: what's the number of people needed so that the probability for at least 2 of them to have the same birthday is more than 50%? The answer is 23, which is quite a small number. This problem is an example to demonstrate that collisions are not unlikely to happen, and thus a good implementation of hash table must be able to resolve collision properly. There are two popular solutions:

Chaining A linked list is kept in each bucket containing items with the correspondent hash value. Given an object x, an insertion / deletion / lookup is executed in the list $A[h(x)]$ when a correspondent operation is executed on the hash table with x.

Open addressing A bucket only stores one object. The hash function specifies a probe sequence $h_1(x), h_2(x)$, etc. When an object is inserted in to the hash table, the sequence will be followed until an empty slot is found. The sequence can be linear (i.e. slots are probed consecutively), or decided by two independent hash functions.

For a hash table with chaining, insertions are always $\Theta(1)$ because we simply insert a new element at the front of a list, while deletions and lookups are $\Theta(list\,length)$. The maximal length of a list can be anywhere from m/n, which means lengths of all lists are equal, to m, which means all objects are in the same bucket. The situation with open addressing is similar. Obviously, the performance of an implementation depends heavily on the choice of the hash function. A good hash function should lead to good performance, i.e. data should be spread out among all hash values, and the result of the hash function should be easy to evaluate and store.

A widely used method to define a hash function consists of two steps. First an object is transformed into a usually large integer, namely the hash code, and then the integer is mapped by a compression function to a number between 0 and $n-1$, i.e. the index of a bucket. The $mod\ n$ function can serve as the compression function.

The number of buckets n must be selected with caution. It should be a prime within a constant factor of the number of objects supposed to be saved in the table, and it should not be close to a power of 2 or 10.

Definition 13. *The load factor α of a hash table is defined as*

$$\alpha = \frac{\#\ of\ objects\ in\ the\ hash\ table}{\#\ of\ buckets\ in\ the\ hash\ table}.$$

Obviously, for open addressing, α has to be smaller than 1, whereas chaining can cope with $\alpha < 1$. In general, α has to be $O(1)$ to guarantee constant running time for hash table operations. In particular, $\alpha \ll 1$ is expected for open addressing.

5.3.3 Universal Hashing

We wish to fabricate a clever hash function that can spread any data set quasi-evenly among all buckets. Unfortunately such function does not exist, because any hash function has a pathological data set. The reason is that for any hash function $h : U \to \{0, 1, \ldots, n-1\}$, according to the Pigeonhole Principle, there exists a bucket i such that at least $|U|/n$ elements of U hash to i under h. If the data set is a subset of these elements, all of them will collide. This could become dangerous in real-world systems: a simple hash function can be reverse engineered and abused.

There are two solutions to this problem. Either a cryptographic hash function, e.g. SHA-2, should be used to make the reverse engineering infeasible, or a randomized approach should be taken: we should design a family H of hash functions such that for any data set S, a randomly chosen function $h \in H$ is almost guaranteed to spread S out quasi-evenly. Such a family of hash functions is called universal.

Definition 14. *Let H be a set of hash functions $h : U \to \{0, 1, \ldots, n-1\}$. H is universal if and only if $\forall x, y \in U (x \neq y)$,*

$$P(h(x) = h(y)) \leq \frac{1}{n},$$

in which n is the number of buckets and h is a hash function chosen uniformly at random from H. $1/n$ is actually the probably of a collision for pure random hashing.

We will now provide a universal hash function family for IP addresses. Let U represent the universe of all IP address of the form (x_1, x_2, x_3, x_4), in which each x_i is an integer between 0 and 255 inclusive. Let n be a prime whose value is comparable with the number of objects in the hash table, and larger than 255. We define a hash function h_a for each 4-tuple $a = (a_1, a_2, a_3, a_4)$ with each $a_i \in \{0, 1, \ldots, n-1\}$:

$$h_a(x_1, x_2, x_3, x_4) = \left(\sum_{i=1}^{4} a_i x_i \right) \mod n.$$

Then the family of all h_a is universal.

Proof. Consider two distinct IP addresses $x = (x_1, x_2, x_3, x_4)$, $y = (y_1, y_2, y_3, y_4)$, and assume that $x_4 \neq y_4$. If x and y collide, we have

$$\left(\sum_{i=1}^{4} a_i x_i \right) \mod n = \left(\sum_{i=1}^{4} a_i y_i \right) \mod n$$

$$a_4(x_4 - y_4) \mod n = \left(\sum_{i=1}^{3} a_i(y_i - x_i)\right) \mod n$$

For an arbitrarily fixed choice of a_1, a_2, a_3, the rhs is a fixed number between 0 and $n-1$ inclusive. With $x_4 - y_4 \mod n \neq 0$ (guaranteed by $n > 255$ and $x_4 \neq y_4$) and a_4 randomly chosen in $\{0, 1, \ldots, n-1\}$, the lhs is actually equally likely to be any of $\{0, 1, \ldots, n-1\}$. Therefore the probability of collision is $\frac{1}{n}$. □

Now we would like to verify the $O(1)$ running time guarantee of hash table implemented with chaining and hash function h selected randomly from a universal family H. Here we assume that $|S| = O(n)$, i.e. $\alpha = \frac{|S|}{n} = O(1)$, and that it takes $O(1)$ to evaluate the hash function.

Proof. As discussed before, the running time of basic operations on a hash table implemented with chaining is $O(list\ length)$. So here we will try to prove that the expectation of the list length L is $O(1)$.

For a specific list corresponding to hash value $h(x)$, we define

$$Z_y = \begin{cases} 1 \text{ if } h(y) = h(x) \\ 0 \text{ otherwise} \end{cases}$$

for any $y \in S$. Then obviously $L = \sum_{y \in S} Z_y$. Thus we have

$$\begin{aligned} E[L] &= \sum_{y \in S} E[Z_y] = \sum_{y \in S} P(h(y) = h(x)) \\ &\leq \sum_{y \in S} \frac{1}{n} = \frac{|S|}{n} = O(1). \end{aligned}$$

□

The running time of operations on hash table implemented with open addressing is hard to analyze. We will use a heuristic assumption that all $n!$ probe sequences are equally possible, which is indeed not true but facilitates an idealized quick analysis. Under this heuristic assumption, the expected running time of operations is $\frac{1}{1-\alpha}$.

Proof. A random probe finds an empty slot with probability $1 - \alpha$. A random probe sequence can be regarded as repetitions of random probes[2]. Thus the expectation of the number of probes needed for finding an empty slot is $\frac{1}{1-\alpha}$. □

[2]Actually the probability for probes after the first probe to find an empty slot is larger, because we don't examine the same slot twice. The running time $\frac{1}{1-\alpha}$ is an upper bound.

For linear probing, the heuristic assumption is deadly wrong. So we assume instead that the initial probe is random, which is again not true in practice. Knuth proved in 1962 that the expected running time of an insertion under this assumption is $\frac{1}{(1-\alpha)^2}$.

5.4 Bloom Filters

List of Algorithms