Notes for Algorithms: Design and Analysis

Jing LI
pkuyplijing@gmail.com

June 28, 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 problems 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 1.1. *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 1.1.

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

[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 1.1. □

Theorem 1.2. *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 1.2. *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 1.3. *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 1.4. *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 1.3. $n^{k-1} = o(n^k), \forall k \geq 1$

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 2.1. *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.

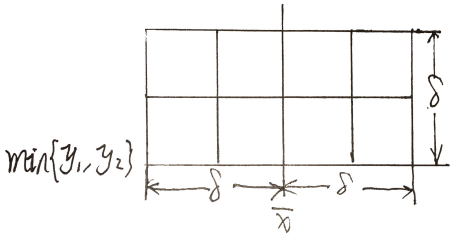

Figure 2.1: p, q at most 7 positions away from each other

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 2.1, 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 2.2.

Theorem 2.2. *$T(n)$ 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 3.1.

Theorem 3.1. *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 3.2.

Lemma 3.2. *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 3.2 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 3.3.

Theorem 3.3. *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 3.1. *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

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

List of Algorithms