CAMBRIDGE

Cambridge University Press
978-1-108-49682-7 — Design and Analysis of Algorithms
Sandeep Sen , Amit Kumar
Excerpt
More Information

1
CHAPTER

Model and Analysis

When we make a claim such as *Algorithm A has running time* $O(n^2 \log n)$, we have an underlying computational model where this statement is valid. It may not be true if we change the model. Before we formalize the notion of a *computational model*, let us consider the example of computing Fibonacci numbers.

1.1 Computing Fibonacci Numbers

One of the most popular sequences is the *Fibonacci* sequence defined by

$$F_i = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ F_{i-1} + F_{i-2} & \text{otherwise for } i \geq 2 \end{cases}$$

It is left as an exercise problem to show that

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \phi'^n) \quad \text{where} \quad \phi = \frac{1+\sqrt{5}}{2} \quad \phi' = 1 - \phi$$

Clearly, it grows exponentially with n and F_n has $\theta(n)$ bits.

Since the closed form solution for F_n involves the *golden ratio* – an irrational number – we must find a way to compute it efficiently without incurring numerical errors or approximations as it is an integer.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49682-7 — Design and Analysis of Algorithms
Sandeep Sen , Amit Kumar
Excerpt
More Information

Method 1

By simply using the recursive formula, one can easily argue that the number of operations (primarily additions) involved is proportional to the value of F_n. We just need to unfold the recursion tree where each internal node corresponds to an addition. As we had noted earlier, this leads to an exponential time algorithm and we cannot afford it.

Method 2

Observe that we only need the last two terms of the series to compute the new term. Hence, by applying the principle of *dynamic programming*,[1] we successively compute F_i starting with $F_0 = 0$ and $F_1 = 1$ and use the previously computed terms, F_i and F_{i-1} for $i \geq 2$.

This takes time that is proportional to approximately n additions, where each addition involves adding (increasingly large) numbers. The size of $F\lceil n/2 \rceil$ is about $n/2$ bits; so, the last $n/2$ computations will take $\Omega(n)$ steps [2] culminating in an $O(n^2)$ algorithm.

Since the nth Fibonacci number is at most n bits, it is reasonable to look for a faster algorithm.

Method 3

$$\begin{bmatrix} F_i \\ F_{i-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{i-1} \\ F_{i-2} \end{bmatrix}$$

By iterating the aforementioned equation, we obtain

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

To compute A^n, where A is a square matrix, we recall the following strategy for recursively computing x^n for a real x and positive integer n.

$$\begin{cases} x^{2k} = (x^k)^2 & \text{for even integral powers} \\ x^{2k+1} = x \cdot x^{2k} & \text{for odd integral powers} \end{cases}$$

We can extend this method to compute A^n.

The number of multiplications taken by the aforementioned approach to compute x^n is bounded by $2 \log n$ (the reader can convince oneself by writing a recurrence). However, the actual running time depends on the time taken to multiply two numbers, which in turn depends on their lengths (number of digits). Let us assume that $M(n)$ is the number of (bit-wise) steps to multiply two n bit numbers. The number of steps to implement the aforementioned approach must take into account the lengths of numbers that are being multiplied. The following observations will be useful.

[1] The reader who is unfamiliar with this technique may refer to a later chapter, Chapter 5, that discusses it in complete detail.

[2] Adding two k bit numbers takes $\Theta(k)$.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49682-7 — Design and Analysis of Algorithms
Sandeep Sen , Amit Kumar
Excerpt
More Information

The length of x^k is bounded by $k \cdot |x|$, where $|x|$ is the length of x.

Therefore, the cost of the the squaring of x^k is bounded by $M(k|x|)$. Similarly, the cost of computing $x \times x^{2k}$ can also be bound by $M(2k|x|)$.The overall recurrence for computing x^n can be written as

$$T_B(n) \leq T_B(\lfloor n/2 \rfloor) + M(n|x|)$$

where $T_B(n)$ is the number of bit operations to compute the nth power using the previous recurrence. The solution of the aforementioned recurrence can be written as the following summation (by unfolding)

$$\sum_{i=1}^{\log n} M(2^i|x|)$$

If $M(2i) > 2M(i)$, then this summation can be bounded by $O(M(n|x|))$, that is, the cost of the last squaring operation.

In our case, A is a 2×2 matrix – each squaring operation involves 8 multiplications and 4 additions involving entries of the matrix. Since multiplications are more expensive than additions, let us count the cost of multiplications only. Here, we have to keep track of the lengths of the entries of the matrix. Observe that if the maximum size of an entry is $|x|$, then the maximum size of an entry after squaring is at most $2|x| + 1$ (Why?). The cost of computing A^n is $O(M(n|x|))$, where the maximum length of any entry is $|x|$ (left as an exercise problem). Hence, the running time of computing F_n using Method 3 is dependent on the multiplication algorithm. Well, multiplication is multiplication – what can we do about it? Before that, let us summarize what we know about it. Multiplying two n digit numbers using the add-and-shift method takes $O(n^2)$ steps, where each step involves multiplying two single digits (bits in the case of binary representation), and generating and managing carries. For binary representation, this takes $O(n)$ steps for multiplying with each bit; finally, n shifted summands are added – the whole process takes $O(n^2)$ steps.

Using such a method of multiplication implies that we cannot do better than $\Omega(n^2)$ steps to compute F_n. For any significant (asymptotically better) improvement, we must find a way to multiply faster.

1.2 Fast Multiplication

Problem Given two numbers A and B in binary, we want to compute the product $A \times B$.

Let us assume that the numbers A and B have lengths equal to $n = 2^k$ – this will keep our calculations simpler without affecting the asymptotic analysis.

$$A \times B = (2^{n/2} \cdot A_1 + A_2) \times (2^{n/2} \cdot B_1 + B_2)$$

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49682-7 — Design and Analysis of Algorithms
Sandeep Sen , Amit Kumar
Excerpt
More Information

where A_1 (B_1) is the leading $n/2$ bits of A (B). Likewise, A_2 is the trailing $n/2$ bits of A. We can expand this product as

$$A_1 \times B_1 \cdot 2^{n/2} + (A_1 \times B_2 + A_2 \times B_1) \cdot 2^{n/2} + A_2 \times B_2$$

Observe that multiplication by 2^k can be easily achieved in binary by adding k trailing 0s (likewise, in any radix r, multiplying by r^k can be done by adding trailing zeros). Hence, the product of two n bit numbers can be achieved by recursively computing four products of $n/2$ bit numbers. Unfortunately, this does not improve things (see exercise 1.6).

We can achieve an improvement by reducing it to three recursive calls of multiplying $n/2$ bit numbers by rewriting the coefficient of $2^{n/2}$ as follows

$$A_1 \times B_2 + A_2 \times B_1 = (A_1 + A_2) \times (B_1 + B_2) - (A_1 \times B_1) - (A_2 \times B_2)$$

Although strictly speaking, $A_1 + A_2$ is not $n/2$ bits but at most $n/2 + 1$ bits (Why?), we can still view this as computing three separate products involving $n/2$ bit numbers recursively and subsequently subtracting appropriate terms to get the required products. Subtraction and additions are identical in modulo arithmetic (2's complement), so the cost of subtraction can be bounded by $O(n)$. (What is the maximum size of the numbers involved in subtraction?). This gives us the following recurrence

$$T_B(n) \leq 3 \cdot T_B(n/2) + O(n)$$

where the last term accounts for addition, subtractions, and shifts. It is left as an exercise problem to show that the solution to this recurrence is $O(n^{\log_2 3})$. This running time is roughly $O(n^{1.7})$, which is asymptotically better than n^2 and therefore we have succeeded in designing an algorithm to compute F_n faster than n^2.

It is possible to multiply much faster using a generalization of the aforementioned method in $O(n \log n \log \log n)$ bit operations utilizing Schonage and Strassen's method. However, this method is quite involved as it uses discrete Fourier transform computation over modulo integer rings and has fairly large constants that neutralize the advantage of the asymptotic improvement unless the numbers are a few thousand bits long. It is, however, conceivable that such methods will become more relevant as we may need to multiply large keys for cryptographic/security requirements. We discuss this algorithm in Chapter 9.

1.3 Model of Computation

Although there are a few thousand variations of the computer with different architectures and internal organization, it is best to think about them at the level of the assembly

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49682-7 — Design and Analysis of Algorithms
Sandeep Sen , Amit Kumar
Excerpt
More Information

language. Despite architectural variations, the assembly level language support is very similar – the major difference being in the number of registers and the word length of the machine. However, these parameters are also in a restricted range of a factor of two, and hence, asymptotically in the same ballpark. In summary, we can consider any computer as a machine that supports a basic instruction set consisting of arithmetic and logical operations and memory accesses (including indirect addressing). We will avoid cumbersome details of the exact instruction set and assume realistically that any instruction of one machine can be simulated using a constant number of available instructions of another machine. Since analysis of algorithms involves counting the number of operations and not the exact timings (which could differ by an order of magnitude), the aforementioned simplification is justified.

The careful reader would have noticed that during our detailed analysis of Method 3 in the previous section, we were not simply counting the number of arithmetic operations but actually the number of bit-level operations. Therefore, the cost of a multiplication or addition was not unity but proportional to the length of the input. Had we only counted the number of multiplications for computing x^n, it would only be $O(\log n)$. This would indeed be the analysis in a *uniform cost* model, where only the number of arithmetic (also logical) operations are counted and the cost does not depend on the length of the operands. A very common use of this model is for comparison-based problems like sorting, selection, merging, and many data-structure operations. For these problems, we often count only the number of comparisons (not even other arithmetic operations) without bothering about the length of the operands involved. In other words, we implicitly assume $O(1)$ cost for any comparison. This is not considered unreasonable since the size of the numbers involved in sorting does not increase during the course of the algorithm for most of the commonly known sorting problems. On the other hand, consider the following problem of repeated squaring n times starting with 2. The resultant is a number 2^{2^n}, which requires 2^n bits to be represented. It will be very unreasonable to assume that a number that is exponentially long can be written out (or even stored) in $O(n)$ time. Therefore, the uniform cost model will not reflect a realistic setting for this problem.

On the other extreme is the *logarithmic* cost model where the cost of an operation is proportional to the length of the operands. This is very consistent with the physical world and is also similar to the *Turing machine* model which is a favorite of complexity theorists. Our analysis in the previous section is actually done with this model in mind. It is not only the arithmetic operations but also the cost of memory access that is proportional to the length of the address and the operand.

The most commonly used model is something in between. We assume that for an input of size n, any operation involving operands of size $\log n$ [3] takes $O(1)$ steps. This is

[3]We can also work with $c \log n$ bits as the asymptotic analysis does not change for a constant c.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49682-7 — Design and Analysis of Algorithms
Sandeep Sen , Amit Kumar
Excerpt
More Information

justified as follows. All microprocessor chips have specialized *hardware circuits* for arithmetic operations like multiplication, addition, division, etc. that take a fixed number of clock cycles when the operands fit into a word. The reason that $\log n$ is a natural choice for a word is that, even to address an input size n, you require $\log n$ bits of address space. The present high-end microprocessor chips have typically 2–4 GBytes of RAM and about 64 bits word size – clearly 2^{64} exceeds 4 GBytes. We will also use this model, popularly known as *random access machine* (or RAM in short), except for problems that deal with numbers as inputs like multiplication in the previous section where we will invoke the *log cost* model. In the beginning, it is desirable that for any algorithm, we get an estimate of the maximum size of the numbers to ensure that operands do not exceed $\Omega(\log n)$ so that it is safe to use the RAM model.

1.4 Randomized Algorithms: A Short Introduction

The conventional definition of an algorithm demands that an algorithm solves a given instance of a problem *correctly* and *certainly*, that is, for any given instance I, an algorithm $\mathcal{A}$ should return the correct output every time without fail. It emphasizes a deterministic behavior that remains immutable across multiple runs. By exploring beyond this conventional boundary, we have some additional flexibility that provides interesting trade-offs between correctness and efficiency, and also between predictability and efficiency. These are now well-established techniques in algorithm design known as *randomized techniques*. In this section, we provide a brief introduction to these alternate paradigms, and in this textbook, we make liberal use of the technique of randomization which has dominated algorithm design in the past three decades leading to some surprising results as well as simpler alternatives to conventional design techniques.

Consider an array A of n elements such that each element is either colored red or green. We want to output an index i, such that $A[i]$ is green. Without any additional information or structure, we may end up inspecting every element of A to find a green element. Suppose we are told that half the elements are colored green and the remaining red. Even then we may be forced to probe $n/2$ elements of the array before we are assured of finding a green element since the first $n/2$ elements that we probe could be all red. This is irrespective of the distribution of the green elements. Once the adversary knows the probe sequence, it can force the algorithm to make $n/2$ probes.

Let us now assume that all $\binom{n}{n/2}$ choices of green elements are equally likely – in what way can we exploit this? With a little reflection, we see that every element is equally likely to be red or green and therefore, the first element that we probe may be green with probability $= 1/2$. If so, we are done – however, it may not be green with probability $1/2$. Then, we can probe the next location and so on until we find a green element. From our earlier argument, we may have to probe at most $n/2$ locations before we succeed. But there

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49682-7 — Design and Analysis of Algorithms
Sandeep Sen , Amit Kumar
Excerpt
More Information

is a crucial difference – it is very *unlikely* that in a random placement of green elements, all the first $n/2$ elements are red. Let us make this more precise.

If the first $m < n/2$ elements are red, it implies that all the green elements got squeezed in the $n-m$ locations. If all placements are equally likely, then the probability of this scenario is

$$\frac{\binom{n-m}{n/2}}{\binom{n}{n/2}} = \frac{(n-m)! \cdot (n/2)!}{n! \cdot (n/2-m)!} = \frac{(n-m)(n-m-1)\cdots(n/2-m+1)}{n(n-1)\cdots(n/2+1)}$$

It is easy to check that this probability is at most $e^{-m/2}$. Therefore, the expected number of probes is at most

$$\sum_{m \geq 0} (m+1) \cdot e^{-m/2} = O(1)$$

In the previous discussion, the calculations were based on the assumption of random placement of green elements. Can we extend it to the general scenario where no such assumption is required? This turns out to be surprisingly simple and obvious once the reader realizes it. The key to this is – instead of probing the array in a pre-determined sequence $A[1], A[2], \ldots$, we probe using a random sequence, say $j_1, j_2, \ldots, j_n$, where $j_1, \ldots, j_n$ is a permutation of $\{1, \ldots, n\}$.

How does this change things ? Since $n/2$ locations are green, a random probe will yield a green element with probability $1/2$. If it is not green, then the subsequent random probes (limited to the unprobed locations) will have even higher probability of the location having a green element. This is a simple consequence of conditional probability given that all the previous probes yielded red elements. To formalize, let X be a random variable that denotes the number of probes made to find the first green element. Then,

$$\Pr[X = k] = \text{The probability that the initial } k-1 \text{ probes are red and the } k\text{-th probe is green}$$
$$\leq 1/2^k$$

The reader must verify the correctness of this expression. The expression can also be modified to yield

$$\Pr[X \geq k] \leq \sum_{i=k}^{i=n/2} 1/2^i \leq 1/2^{k-1},$$

and the expected number of probes is at most $O(1)$.

This implies that the number of probes not only decreases exponentially with k but is *independent of the placement of the green elements*, that is, the worst-case scenario is over all possible input arrays. Instead of relying on the randomness of the placement (which is not in our control), the algorithm itself uses a random probe sequence matching the same phenomenon. This is the essence of a *randomized* algorithm. In this case, the final result is

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49682-7 — Design and Analysis of Algorithms
Sandeep Sen , Amit Kumar
Excerpt
More Information

always correct, that is, a green element is output but the running time (number of probes) is a random variable and there is a trade-off between the number of probes k and the probability of termination within k probes.

If the somewhat hypothetical problem of finding a green element from a set of elements has not been convincing in terms of its utility, here is a classical application of the aforementioned solution. Recall the quicksort sorting algorithm. In quicksort, we partition a given set of n numbers around a pivot. It is known that the efficiency of the algorithm depends primarily on the relative sizes of the partition – the more balanced they are in size, the better. Ideally, one would like the pivot to be the median element so that both sides of the partition are small. Finding the median element is a problem in itself; however, any element around the median is almost equally effective, say an element with rank[4] between $[\frac{n}{4}, \frac{3n}{4}]$ will also lead to a balanced partitioning. These $n/2$ elements can be thought of as the green elements and so we can apply our prior technique. There is a slight catch – how do we know that the element is green or red? For this, we need to actually compute the rank of the probed element, which takes $n-1$ comparisons but this is acceptable since the partitioning step in quicksort takes n steps and will subsume this. However, this is not a complete analysis of quicksort which is a recursive algorithm; we require more care that will be discussed in a later chapter dealing with selections.

1.4.1 A different flavor of randomized algorithms

Consider a slight twist on the problem of computing the product of two $n \times n$ matrices $C = A \times B$. We are actually given A, B, C and we have to verify if C is indeed the product of the two matrices A and B. We may be tempted to actually compute $A \times B$ and verify it element by element with C. In other words, let $D = A \times B$ and we check if $C - D = \mathbb{O}^n$, where the right-hand side is an $n \times n$ matrix whose elements are identically 0.

This is a straightforward and simple algorithm, except that we will pay the price for computing the product which is not really necessary for the problem. Using elementary method for computing matrix products, we will need about $O(n^3)$ multiplications and additions[5], whereas an ideal algorithm could be $O(n^2)$ steps, which is the size of the input. To further simplify the problem and reduce dependency on the size of each element, let us consider Boolean matrices and review addition modulo 2. Examine the algorithm described in Figure 1.1. It computes three matrix vector products – BX, $A(BX)$, and CX–incurring a total of $3n^2$ operations which matches the input size and therefore, is optimal.

[4]The rank of x is the number of elements in the set smaller than x.
[5]There are sophisticated and complex methods to reduce the number of multiplications below n^3 but they are still much more than n^2.

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49682-7 — Design and Analysis of Algorithms
Sandeep Sen , Amit Kumar
Excerpt
More Information

```
Procedure Verifying matrix product($A, B, C$)
1 Input: $A, B, C$ are $n \times n$ matrices over GF(2);
2 Output: If $A \cdot B = C$ then Yes else No;
3 Choose a random 0–1 vector $X$;
4 if $A \cdot (B \cdot X) = C \cdot X$ then
5     Return YES;
6 else
7     Return NO
```

Figure 1.1 *Algorithm for verifying matrix product*

Observation If $A(BX) \neq CX$, then $AB \neq C$.

However, the converse, that is, $A(BX) = C \implies AB = C$ is not easy to see. On the contrary, consider the following example, which raises serious concerns.

Example 1.1 $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ $B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ $C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} AB = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

$$X = \begin{bmatrix} 1 \\ 0 \end{bmatrix} ABX = \begin{bmatrix} 1 \\ 0 \end{bmatrix} CX = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$X' = \begin{bmatrix} 0 \\ 1 \end{bmatrix} ABX' = \begin{bmatrix} 1 \\ 0 \end{bmatrix} CX' = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Clearly, the algorithm is not correct if we choose the first vector. Instead of giving up on this approach, let us get a better understanding of the behavior of this simple algorithm.

Claim 1.1 *For an arbitrary vector (non-zero) Y and a random vector X, the probability that the dot product $X \cdot Y = 0$ is less than 1/2.*

There must be at least one $Y_i \neq 0$ – choose that X_i last; with probability $1/2$, it will be non-zero. For the overall behavior of the algorithm, we can claim the following.

Claim 1.2 *If $A(BX) \neq CX$, then $AB \neq C$, that is, the algorithm is always correct if it answers NO. When the algorithm answers YES, then* $\Pr[AB = C] \geq 1/2$.

If $AB \neq C$, then in $AB - C$, at least one of the rows is non-zero and from the previous claim, the dot product of a non-zero vector with a random vector is non-zero with probability $1/2$. It also follows that by repeating this test and choosing independently another random vector when it returns YES, we can improve the probability of success and our confidence in the result. If the algorithm returns k consecutive YES, then $\Pr[AB \neq C] \leq \frac{1}{2^k}$.

The reader may have noted that the two given examples of randomized algorithms have distinct properties. In the first example, the answer is always correct but the running

© in this web service Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-1-108-49682-7 — Design and Analysis of Algorithms
Sandeep Sen , Amit Kumar
Excerpt
More Information

time has a probability distribution. In the latter, the running time is fixed, but the answer may be incorrect with some probability. The former is known as *Las Vegas* and the latter is referred to as *Monte Carlo* randomized algorithm. Although in this particular example, the Monte Carlo algorithm exhibits asymmetric behavior (it can be incorrect only when the answer is YES), it need not be so.

1.5 Other Computational Models

There is clear trade-off between the simplicity and the fidelity achieved by an abstract model. One of the obvious (and sometimes serious) drawbacks of the RAM model is the assumption of unbounded number of registers since the memory access cost is uniform. In reality, there is a memory hierarchy comprising registers, several levels of cache, main memory, and finally the disks. We incur a higher access cost as we go from registers toward disks and for technological reasons, the size of the faster memory is limited. There could be a disparity of 10^5 between the fastest and the slowest memory which makes the RAM model somewhat suspect for larger input sizes. This has been redressed by the *external memory model*.

1.5.1 External memory model

In this model, the primary concern is the number of disk accesses. Given the rather high cost of a disk access compared to any CPU operation, this model actually ignores all other costs and counts only the number of disk accesses. The disk is accessed as contiguous memory locations called *blocks*. The blocks have a fixed size B and the simplest model is parameterized by B and the size of the faster memory M. In this two-level model, the algorithms are only charged for transferring a block between the internal and external memory; all other computations are free. The cost of sorting n elements is $O\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$ disk accesses and this is also optimal. To see this, we can analyze M/B-way merge sort in this model. Note that one block from each of the M/B sorted streams can fit into the main memory. Using appropriate data structures, we can generate the next B elements of the output and we can write an entire block to the output stream. Hence, the overall number of I-Os per phase is $O(n/B)$ since each block is read and written exactly once. The algorithm makes $O(\frac{n/B}{M/B})$ passes, yielding the required bound.

There are further refinements to this model that parameterizes multiple levels and also accounts for internal computation. As the model becomes more complicated, designing algorithms also becomes more challenging and often more laborious. We discuss algorithm design and analysis in this model and many variations in Chapter 15.

© in this web service Cambridge University Press www.cambridge.org