



Chapter 2 - Warming Up: Recursion, Divide and Conquer

Data Structures & Algorithms (University of Houston)

2

Warming up: Recursion, Divide and Conquer

In this chapter we discuss some basic concepts in algorithm design, especially, recursion, divide and conquer, and algorithm analysis.

What is an Algorithm ? An algorithm is a sequence of **precise** instructions that (usually) **terminates**. “Precise” means that the instructions should be unambiguous and mathematically rigorous. Termination criterion ensures that the algorithm finishes in finite time. Although most algorithms terminate in a finite number of steps, there are many useful algorithms that go on forever (can you think of some example ?). We will generally deal with algorithms that have a finite running time.

2.1 Two Warmup Problems

Let’s warmup by looking at some algorithmic puzzle-like problems that also illustrate a key algorithmic technique.

2.1.1 Find the Celebrity

Problem 2.1. A **celebrity** among a group of n people is a person who **knows nobody** but is **known by everybody else**. The task is to identify a celebrity by only asking questions to people of the following form: “Do you know this person ?”. Design an “efficient” algorithm to identify a celebrity or determine that a group has no such person. What is the largest number of questions your algorithm needs to solve this problem?

Solution This problem can be solved by reducing the problem size by one — **decrease and conquer** strategy, a **recursive** technique that is useful in solving many problems. In the decrease and conquer strategy we reduce the problem size by one (or sometimes by more than one) and then recursively apply the same strategy to the reduced problem till the problem size becomes small (e.g., 1). Let's illustrate this by solving the celebrity problem.

We start with n persons. Choose 2 persons, say A and B and ask A whether he/she knows B. If yes, then A cannot be a celebrity and can be eliminated. If no, then B cannot be a celebrity and can be eliminated. This reduces the problem size by one. We repeat the same strategy in among the group of $n - 1$ persons.

After $n - 1$ questions, one person (say X) will remain. If there is a celebrity then it should be this person (why ?) To check whether this person is a celebrity we ask X whether he knows everybody else and whether everybody else knows X. Total number of questions = $n - 1 + n - 1 + n - 1 = 3n - 3$. (Actually, we can improve this to $3n - 4$. How?)

A even better algorithm can be given! It can be shown that it can be done using $3n - \lfloor \log n \rfloor - 3$ questions and this is the best possible!, i.e., in general, one cannot solve this problem using a lesser number of questions. (Throughout the book, unless otherwise stated, by log, we will mean logarithm to the base 2.)

Note that the performance or “complexity” in the above problem is naturally determined by the number of questions. The complexity of an algorithm is typically determined by the costliest operation that dominates the overall cost of the algorithm.

Exercise 2.1. Show that the celebrity problem can be solved by using $3n - \log n - 3$ questions (assume that n is a power of 2). (Hint: Group people into pairs — similar to a knockout tournament, e.g., Wimbledon tennis tournament.)

2.1.2 Tiling a Board

Problem 2.2. You are given a $2^n \times 2^n$ board missing one square. The missing square can be any of the board squares. See figure 2.1. Cover all the squares (except the missing square) by L-shaped tiles exactly with no overlaps.

Solution The original problem gives rise to 4 subproblems of the **same kind**. That is each of the subproblems is exactly of the same type as the

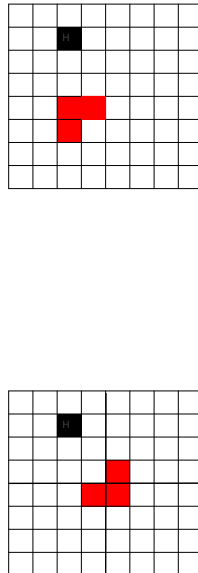


Figure 2.1: Tiling problem (top) and its solution (bottom)

original problem — its dimensions are a power of 2 (in fact, reduced by half) and it has one missing square. This strategy is called **divide and conquer**, a very powerful technique. It can be naturally implemented using **recursion**. Recursion is central to algorithmic thinking. In fact, we will emphasize on writing algorithms using recursion as much as possible (unless non-recursive algorithms are much easier to write). Typically, recursive algorithms are conceptually simpler to understand and easier to write.

It is important to note that in decrease/divide and conquer strategies that the resulting subproblems should be of the **same kind, but strictly smaller in size**. It is even better if the subproblem sizes go down by a constant factor, in which case the number of “recursive levels” (i.e., the depth of the recursion) will be **logarithmic** in the problem size.

Exercise 2.2. *Compute the number of L-shaped tiles needed to cover a $2^n \times 2^n$ board. Show that it is $(4^n - 1)/3$. Obtain the same answer by setting up a recurrence for $T(n)$, which is the number of tiles needed to cover a*

$2^n \times 2^n$ board.

2.2 Problem: Prime Number (Primality) Checking

Let's now look at a classical real-world problem, namely, primality checking, whose complete solution was given relatively recently (in the last decade), though the problem itself has been around for a long time.

Problem 2.3 (Primality checking). *Given a natural number, say, n , is it a prime number or not?*

This is a problem that requires just a boolean answer: yes or no. It is central to number theory and also to algorithms. Later in the course we will give an efficient (randomized) algorithm for this problem, but first let's look at some obvious algorithms.

2.2.1 Pseudocode for Algorithm IsPrime

Algorithm 1 Algorithm **IsPrime**(n). Input: A natural number n . **Output:** Yes (if n is prime) or No (if n is composite).

```

1: for  $i := 2$  to  $n - 1$  do
2:   if  $i$  divides  $n$  then
3:     Output "No"
4:   Halt
5: Output "Yes"

```

2.2.2 Correctness of the Algorithm IsPrime

Why is the Algorithm **IsPrime** correct?

The correctness follows because it correctly implements the mathematical definition of a prime number. The **for** loop checks for divisibility of n by **every** number between 2 and $n - 1$.

2.2.3 Speeding up the algorithm

Algorithm **IsPrime** is rather slow, especially if the input number n is **prime**. In this case, the **for** loop runs $n - 2$ times. Thus the running time of the algorithm is **approximately proportional** to n , the **magnitude** of the input number.

Can we speed up **IsPrime**? For this, we use the following simple fact.

Fact 2.1. *If i is a divisor of a number n , then n/i is also a divisor of n .*

We need to check for divisors only till $\max_{2 \leq i \leq n-1}(\min\{i, n/i\})$. The maximum (among all i , i taking values from 2 to $n - 1$) is attained when $i = n/i$, i.e., $i^2 = n$ or $i = \sqrt{n}$. Hence it is enough to check for divisors up to $\sqrt{n}$.

2.2.4 FastIsPrime Algorithm

Algorithm 2 Algorithm **FastIsPrime**(n). **Input:** An natural number n . **Output:** Yes (if n is prime) or No (if n is composite).

```

1: for  $i := 2$  to  $\lfloor \sqrt{n} \rfloor$  do
2:   if  $i$  divides  $n$  then
3:     Output “No”
4:   Halt
5: Output “Yes”

```

The run time of **FastIsPrime** is proportional (approximately) to $\sqrt{n}$ which is significantly smaller than n .

However, note that **FastIsPrime** is still quite slow as its running time is an **exponential** function in the **size of the input** (why?). An important aspect of **efficient** algorithms is that their run time scales **polynomially** in the input size. Even with regards to polynomial, we ideally would like to keep the degree of the polynomial as small as possible, e.g., linear time algorithms are great, since their run time is proportional to the size of the input. Such algorithms are very helpful in handling “Big Data” applications, where the input sizes are very very large.

Before we go into the analyzing the performance of algorithms, we need a model for computation, which we state next.

2.3 A Model for (Sequential) Computation: RAM

We describe a simple computation model used to **analyze running time of algorithms**. This model captures computation that takes place in a single processor machine, where instructions are executed one after the other in a sequential manner.

RAM - Random Access Machine - model has the following key features:

1. A computing unit plus a memory.

2. Instructions are executed sequentially (i.e., one after the other, according to the order of the program).
3. Accessing (reading/writing) any one memory cell takes one step (one time unit). This is why it is called “random” access (although there is no randomness). It merely means that regardless of the location, any memory cell (read piece of data stored in memory) can be accessed (read/write) in one time step.
4. Any “reasonable” one operation (e.g., arithmetic operations, comparison etc.) takes one step. It is important to clarify what a reasonable operation means. For example, adding/multiplying/subtracting/dividing two numbers consisting of small number of bits (e.g., 32-bit or 64-bit) is reasonable. However, doing the same for n -bit numbers is not — this will take about n steps. Thus, one has to be careful in accounting for the cost of the underlying primitive operation. Note that in many algorithms, this abstraction of using unit cost for one primitive operation simplifies analysis tremendously, without which it is hard to quantify the performance of the algorithm in a clean way. This is indeed a powerful abstraction!

2.4 Asymptotic Notation: Big-O (O) notation, Big- Ω notation, and Big- Θ notation

Algorithmic analysis will not be as elegant as it is if not due to the fact that we emphasise on asymptotic and approximate analysis, rather than exact analysis for a given input size. In other words, we mainly care about how the algorithm’s performance **scales** with the problem size in a **reasonably approximate** manner that is approximate up to a constant factor (i.e., a constant that is independent of the problem size). This allows the algorithm analyzer to focus on quantifying the dominant function that determines the algorithm’s performance, while avoiding cumbersome details. We will see examples of this philosophy throughout the course. As an example, we will say that `IsPrime` takes $O(n)$ time (assuming that all arithmetic operations involving n -bit numbers take constant time !) and `FastIsPrime` takes $O(\sqrt{n})$ time. We will see later that a better way of expressing these runtimes is with respect to the **size** of the input, which is $\log n$ bits (not n).

Consider two positive functions $g(n)$ and $f(n)$ on natural numbers. Think of n as the **input size** of the algorithm; and f (and g) as denoting

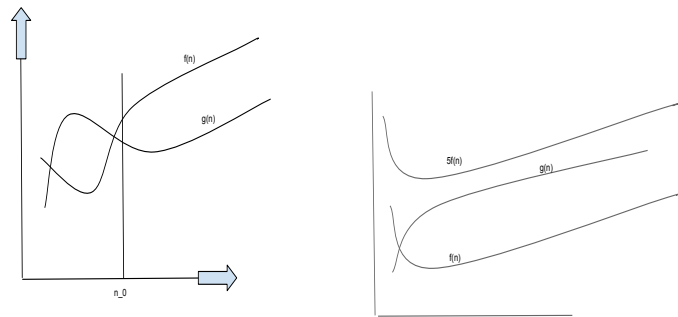


Figure 2.2: Illustration of the Big-O notation. $g(n) = O(f(n))$.

the **runtime** (of an algorithm) as a function of the input size.

Big-O notation A function $g(n)$ is “Big-O” of $f(n)$, i.e., $g(n) = O(f(n))$, if there is some positive constant $c > 0$ (independent of n), such that $cf(n)$ grows asymptotically (i.e., for large n) at least as “fast” as $g(n)$. In other words, $g(n) \leq cf(n)$, for large n , i.e., for all $n \geq n_0$ (n_0 is some fixed number).

Using limits the above can be expressed as:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \leq c.$$

where $c > 0$ is a positive constant.

The above definition assumes that the limit exists (otherwise we can modify the definition to take the $\limsup$ — see examples).

Big- $\Omega(n)$ notation A function $g(n)$ is “Big- $\Omega(n)$ ” of $f(n)$, i.e., $g(n) = \Omega(f(n))$, if there is a positive constant $c > 0$ (independent of n), such that for any $n \geq n_0$, $g(n) \geq cf(n)$.

Using limits the above can be expressed as:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \geq c.$$

The above definition assumes that the limit exists (otherwise we can modify the definition to take the $\liminf$).

Big-Theta (Θ) Notation Two positive functions $g(n)$ and $f(n)$ are said to be "Big-Theta" of each other if their growth is asymptotically **within a constant factor of each other**. That is there are some fixed positive constants c_1 and c_2 such that, $c_1 f(n) \leq g(n) \leq c_2 f(n)$, for all $n > n_0$. Equivalently, $g(n) = \Theta(f(n))$ if $g(n) = O(f(n))$ AND $g(n) = \Omega(f(n))$.

Using limits the above can be expressed as:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = c$$

where c is a positive constant (> 0).

Example 2.1. • $2n = O(n) = \Theta(n)$. Constants are ignored! Big-O, Big-Theta, and Big-Theta notation "hides" **constant factors**.

• $n^3 + 2^{1000}n^2 + 2^{100000}n = O(n^3) = \Theta(n^3)$.

• In general for any polynomial

$$p(n) = \sum_{i=0}^d a_i n^i$$

where a_i are constants and $a_d > 0$, we have $p(n) = \Theta(n^d)$.

• $n = O(n^2)$, but $n \neq \Theta(n^2)$. That is, n^2 grows "strictly" faster than n .

• $n^{1000} = O(2^n)$, but $n^{1000} \neq \Theta(2^n)$.

• **IsPrime** algorithm runs in $O(n)$ time, where n is the (magnitude of the) input number.

•

$$f(n) = \begin{cases} n^2 & : n \text{ is even} \\ n^3 & : n \text{ is odd} \end{cases}$$

$$g(n) = n^3$$

$$f(n) = O(g(n))$$

$$\limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq 1.$$

2.4.1 Little-o and Little- ω notations

Sometimes, we would like asymptotic notation to capture a function that grows **strictly slower** than some other function. Given two functions, $g(n)$

2.4. ASYMPTOTIC NOTATION: BIG-O (O) NOTATION, BIG- Ω NOTATION, AND BIG- Θ NOTATION 15

and $f(n)$, we say that $g(n) = o(f(n))$ if for **any** positive constant c there is a constant n_c such that for any $n > n_c$, $g(n) < cf(n)$.

Alternatively, the above can be expressed as

$$\limsup_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0.$$

Similarly, we would like asymptotic notation to capture a function that grows **strictly faster** than some other function. Given two functions, $g(n)$ and $f(n)$, we say that $g(n) = \omega(f(n))$ if for any positive constant c there is a constant n_c such that for any $n > n_c$, $g(n) > cf(n)$.

Alternatively, the above can be expressed as

$$\liminf_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \infty.$$

Example 2.2. • $1/n = o(1)$

- $n^{1000} = o(1.1^n)$
- $n^2 = \omega(n)$
- $n = (?)n^{1+\sin n}$

$$\limsup_{n \rightarrow \infty} \frac{n}{n^{1+\sin n}} = \infty$$

$$\liminf_{n \rightarrow \infty} \frac{n}{n^{1+\sin n}} = 0$$

Hence the two functions are not asymptotically comparable via little- o or little- ω notation.

2.4.2 Some common runtimes of algorithms

We use the following terminology for some common run times of algorithms.

- $O(n)$ — Linear time.
- $O(n^2)$ — Quadratic time.
- $O(n^c)$ (for some constant $c > 0$) — Polynomial time.
- $O(2^n)$ or $O(c^n)$ (for some constant $c > 0$) — Exponential time.
- $O(\log n)$ — Logarithmic time.

Polynomial time algorithms are considered “efficient” — typically preferred compared to exponential time algorithms. This although has to be carefully interpreted, especially in practice. For example, a n^{10} algorithm may not be preferable over a $(1.1)^n$ algorithm (although the latter is exponential), for even reasonably large-size values of n . However, a phenomenon that has been seen quite often in the theory of algorithms is that, once polynomial time solvability is established, the degree of the polynomial can be reduced, leading to more efficient algorithms. Hence breaching the exponential barrier has important consequences both in theory and in practice. Among polynomial time algorithms, we prefer ones with the smallest exponent c . For many “big data” problems with large input size, linear time algorithms are preferred.

Another key point that one should note about asymptotic notation is that it “hides” constants. It is quite possible that an $O(n)$ algorithm can be very impractical since the constant factor masked by the big-O notation can be very large. Such an algorithm might be worse compared to even to an $O(n^2)$ algorithm with a small hidden constant factor. These issues are important when implementing an algorithm in practice.

How do you classify *IsPrime*? Is it a polynomial time algorithm? The input size consists of $\log n$ bits. The run time is $O(n)$. Hence, *IsPrime* is an exponential time algorithm.

How do you classify *FastIsPrime*? Note that *FastIsPrime* algorithm runs in $O(\sqrt{n})$ time (assuming $\sqrt{n}$ can be calculated in constant time!) Hence *FastIsPrime* runs in $O(2^{(\log n)/2}) = O(\sqrt{2^{\log n}})$.

2.5 Recursion

Recursion is a fundamental paradigm in algorithm design. A **recursive** algorithm solves a problem by solving **one or more smaller instances** of the **same** problem. A recursive algorithm will always have a **termination condition**, otherwise the function will keep calling itself forever.

A **recursive function** is one that invokes **itself**. In principle, any algorithm can be expressed using recursive functions — in many cases, the recursive version is much simpler to write. In particular, divide/decrease and conquer algorithms can be naturally implemented using recursion. We will next look at several problems that illustrate recursion and divide and conquer technique.

2.6 Problem: Greatest common divisor (gcd)

Consider the gcd computation problem which is a central to many algorithmic applications involving numbers.

Problem 2.4. *Given two integers a and b find the greatest common divisor (gcd) of a and b , denoted by $\text{gcd}(a, b)$.*

We will design a **recursive** algorithm for finding a gcd of two given numbers. We use the following mathematical fact.

Fact 2.2. *The gcd of two integers a and b with $a > b$ is equal to the gcd of b and $a \bmod b$ (the remainder when a is divided by b). In other words, $\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$.*

Proof. Assuming $a > b$, we can write:

$$a = b * d + (a \bmod b)$$

where d is an integer (the quotient). Hence any number x that divides both a and b should divide $a \bmod b$ as well. $\square$

2.6.1 Euclid's algorithm to compute gcd

Fact 2.2 suggests an algorithm to calculate gcd **recursively** which is attributed to Euclid, dating back over 2000 years and is one of the the oldest algorithms known!

Algorithm 3 Algorithm $\text{gcd}(a, b)$. **Input:** Integers a and b , with $a > b$.
Output: $\text{gcd}(a, b)$.

```

1: if  $b = 0$  then
    // Termination condition
2:   Output  $a$ 
3: Output  $\text{gcd}(b, a \bmod b)$  // Recursive call

```

2.6.2 Recursive function: Two key features

1. **Termination condition:** A recursive algorithm should have one or more termination condition(s).
2. **Recursive call:** It should call **itself** with a **strictly smaller** value of its arguments.

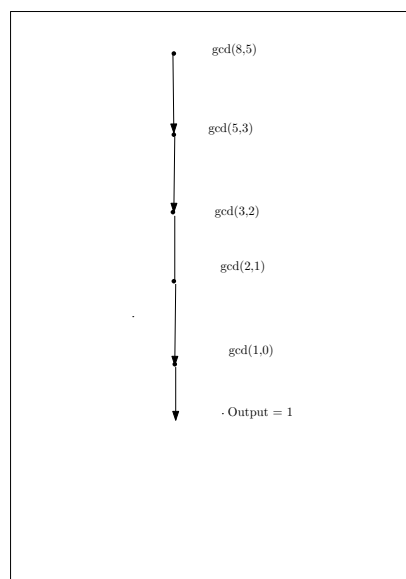


Figure 2.3: Recursion flow diagram for gcd.

The recursion flow diagram Consider invoking Algorithm **gcd** on inputs $a = 8$ and $b = 5$. The sequence of recursive calls can be represented as a recursion flow diagram which, in this case, will be just a **path** (Figure 2.3).

2.6.3 Termination, Correctness, and Running Time

Note that without a termination condition, the recursive algorithm may go on forever. The termination condition is sometimes called as a “base (or basis) case” whose correctness is easy to check. In Algorithm **gcd**, it is clear that if $b = 0$, then a is the gcd (trivially). The algorithm’s correctness follows from repeated application of Fact 2.2 and by the termination condition.

How long does Algorithm **gcd** take? Running time is determined by the **number of recursive calls**. Given inputs a, b (assume $a \geq b > 0$), it can be shown that the **number of recursive calls** is at most $1 + 2\lceil \log_2(b) \rceil$. Thus the algorithm’s running time is $O(\log b)$ (the logarithm of the smaller of the two numbers). That is the running time is essentially proportional to the **number of bits** of b .

Theorem 2.1. *The number of recursive calls generated by Algorithm **gcd** when invoked on inputs a and b ($a \geq b$) is at most $1 + 2\lceil \log_2(b) \rceil$.*

Proof: The first recursive call gives $\text{gcd}(b, a \bmod b)$. Henceforth, we

show that the first argument in the call decreases by at least a factor of 2 in **at most two** recursive calls.

Consider 2 cases:

- Case 1: $a \bmod b \leq b/2$. The next call will be $\text{gcd}(a \bmod b, b \bmod (a \bmod b))$. Hence, the first argument is $\leq b/2$ by assumption.
- Case 2: $a \bmod b > b/2$. The next call will be $\text{gcd}(a \bmod b, b \bmod (a \bmod b))$. In the subsequent call, the first argument will be $b \bmod (a \bmod b)$ which is less than $b/2$, since $a \bmod b > b/2$. Thus in two calls the first argument has reduced by at least a factor of 2.

Thus, after at most every two calls we have the following changes to the first argument:

$$b \rightarrow (< b/2) \rightarrow (< b/4) \rightarrow (< b/8) \rightarrow 1$$

How many “steps” before b reaches 1? Let the number of steps be t . Then $2^t \geq b$, hence $t \geq \log_2(b)$. Thus the number of steps is $\lceil \log_2(b) \rceil$. Hence the overall number of recursive calls is at most $1 + 2\lceil \log_2(b) \rceil = O(\log b)$.

2.7 Problem: Searching for the maximum element

Let's consider another classic problem to illustrate the idea of recursion and its analysis.

Problem 2.5. *Input:* An array $S[1 \dots n]$ of n numbers. *Output:* The maximum number in S .

We can think of an array as an **ordered set (or list)** of elements.

2.7.1 A non-recursive algorithm

This algorithm simply compares each element with the current maximum in a sequential manner. This is called **sequential search**.

Input: Array $S[1 \dots n]$ of size $n \geq 1$.

Output: maximum element in S .

Algorithm $\text{Max}(S[1 \dots n])$

In each step of the **for** loop the current max is compared with the i th element of S ; if it is less then variable **max** is updated. Clearly, at the end of the **for** loop, **max** contains the maximum number in S . Thus, algorithm **Max** takes $O(n)$ comparisons (we will assume that comparison

```

1:  $max = S[1]$  // initialize max to be the first element.
2: for  $i = 1$  to  $n$  do
3:   if  $max < S[i]$  then
4:      $max = S[i]$ 
5: Output max

```

is the costly operation for this algorithm and hence this is a good measure of the running time. Nevertheless, in this example, even if you take the other operations into account, e.g., assignment etc., the overall time is $O(n)$.

2.7.2 A recursive algorithm: Divide and conquer strategy

There are many ways to design a recursive algorithm for finding the maximum element. One can adopt a decrease and conquer strategy whereby the size of the array (in the subproblem) is reduced by one in iteration. This strategy gives rise to one recursive subproblem (Can you see how?).

Alternatively, we adopt a divide and conquer strategy has typically 3 parts: **Split**, **Solve (recursively)**, and **Combine**.

- **Split** the problem into several **smaller independent** subproblems.
- **Recursively solve** the subproblems.
- **Combine** the solutions of the subproblems to get the solution for the original problem.

Divide and conquer strategy for finding max:

- **Split**: Split the given set S into two equal parts (first half and second half).
- **Solve**: Recursively find the maximum of the first part and the second part — let these be max_1 and max_2 respectively.
- **Combine**: Output the maximum of S as the maximum of max_1 and max_2 .

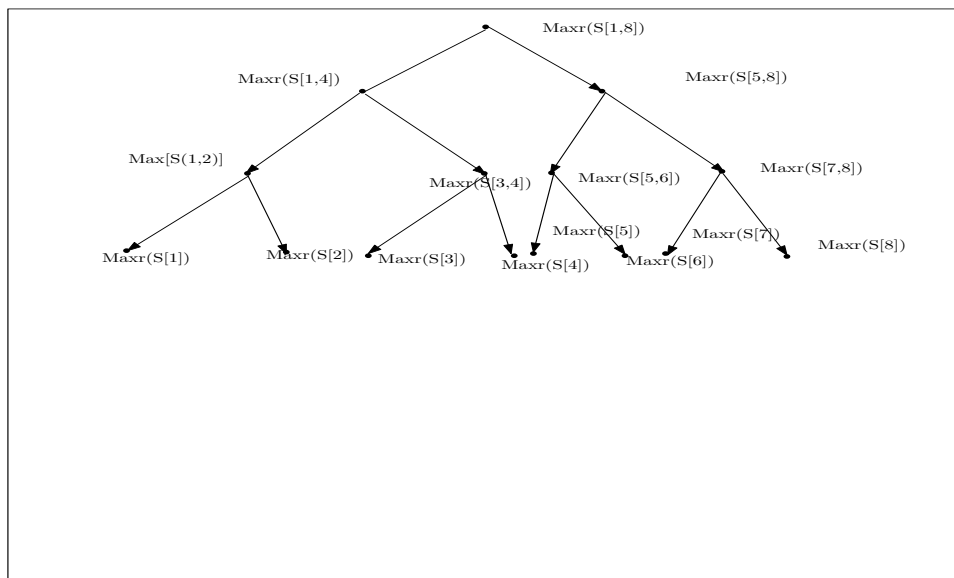
A Divide and Conquer Algorithm To solve the original problem of finding the maximum element in $S[1 \dots n]$, we call $\text{Maxr}(S[1 \dots n])$.

Algorithm 4 Algorithm **Maxr**($S[a \dots b]$). **Input:** Array $S[a \dots b]$, with left index a and right index b . **Output:** Maximum element in S .

```

1: if  $a = b$  then
2:   Return  $S[a]$ 
3:  $m = (a + b)/2$  // middle point
4:  $max_1 = \text{Maxr}(S[a \dots m])$  // first part
5:  $max_2 = \text{Maxr}(S[m + 1 \dots b])$  // second part
6: if  $max_1 > max_2$  then
7:   Return  $max_1$ 
8: else
9:   Return  $max_2$ 

```



The recursion “flow diagram” Consider invoking Algorithm **Maxr** on input $S[1 \dots 8]$.

The sequence of recursive calls can be represented as a “flow diagram” which will be a **tree**. This is usually called as a **Recursion Tree**.

2.7.3 Correctness of Algorithm **Maxr**

It is easy to establish the correctness of the above algorithm by using **mathematical induction**. The induction is on the **size** of the input set S .

This is done in two steps.

Proof by Mathematical Induction

1. **Base case:** This is the trivial case when the size of S is 1. Clearly, the algorithm is correct, since it outputs this single element.
2. **Induction step:** Assume that the algorithm is correct for **all** sets of size $< n$. This is called the **Induction hypothesis**.

Establishing the Induction Step Assuming the induction hypothesis, we will show that the algorithm is correct also for input set of size n , i.e., we will show that $\text{Maxr}(S[1 \dots n])$ is correct. By the induction hypothesis, max_1 and max_2 are correct maximum values of the sets $S[a \dots m]$ and $(S[m+1, b])$ respectively (since they are of size $< n$). Hence the maximum of max_1 and max_2 should be the maximum of the set $S[1 \dots n]$.

Running Time How long does Algorithm **Maxr** take? Running time is determined by the **total number of recursive calls**. The number of recursive calls is equal to the **size of the recursion tree**, i.e., the **number of nodes in the tree**.

In our example, when input is $S[1 \dots 8]$, the size of the tree is $15 = 1 + 2 + 4 + 8 = 2^0 + 2^1 + 2^2 + 2^3 = 2^4 - 1$. Generalizing the above: when the input is $S[1 \dots 2^k]$, the size of the tree $= 2^0 + 2^1 + 2^2 + \dots + 2^k = 2^{k+1} - 1 \leq 2(2^k) = 2n = O(n)$, where $n = 2^k$ is the size of the array. Thus the running time is **proportional to the size of the array**, i.e., $O(n)$.

Analyzing using a Recurrence Recurrence relations are a powerful way to analyze recursive algorithms. This is especially the case with divide/decrease and conquer algorithms. Let's analyze Algorithm **Maxr** using a recurrence. Instead of counting recursive calls, we will count the number of comparisons. Both are equivalent ways (asymptotically) to bound the run time.

Let the function $T(n)$ denote the number of comparisons required by **Maxr** on an input array of size n .

Then we can write the recurrence:

$$T(n) = 2T(n/2) + 1.$$

We also need a “base” case:

$$T(1) = 0.$$

How to solve such a recurrence?

Unwinding the recurrence For convenience, we will assume that n is a power of 2, i.e., $n = 2^k$, for some k . This assumption will not change the

asymptotic result. (Why?) Then:

$$\begin{aligned}
 T(2^k) &= 2T(2^{k-1}) + 1 = 2(2T(2^{k-2}) + 1) + 1 = \\
 &= 2^2T(2^{k-2}) + 2 + 1 \\
 &= 2^3T(2^{k-3}) + 2^2 + 2 + 1 = 2^kT(1) + 2^{k-1} + \dots + 1 \\
 &= 2^{k-1} + 2^{k-2} + 2 + 1 = \sum_{i=0}^{k-1} 2^i.
 \end{aligned}$$

The above sum is obtained by using the formula for the sum of a **geometric** series:

$$\sum_{i=0}^{k-1} x^i = 1 + x + \dots + x^{k-1} = \frac{x^k - 1}{x - 1}.$$

where, x is called the (common) ratio.

Hence,

$$T(n) = \sum_{i=0}^{k-1} 2^i = 2^k - 1.$$

Hence,

$$T(2^k) = 2^k - 1 = n - 1 = n - 1 = O(n).$$

2.8 Techniques for Solving Recurrences

2.8.1 Guess and Verify

Alternative to the “unwinding” technique discussed above, another powerful way to solve recurrences is the “**guess and verify**” method. This uses **mathematical induction** to guess the correct solution to the recurrence and then prove that it is indeed the case, by using induction. Let’s solve the recurrence for Algorithm **Maxr** using this method.

The “real” recurrence is:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

.

Guess $T(n) = O(n)$. Let’s use **induction** to show that $T(n) \leq cn$, for some (suitable) constant c .

Induction hypothesis: Assume that $T(k) \leq ck$ for all $k < n$. Plugging

the hypothesis in the recurrence gives:

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1. \\ &= cn + 1, \end{aligned}$$

which does not prove our hypothesis (Why?)

Revising our Guess We revise our guess to: $T(n) \leq cn - b$ and show this by induction. **Induction hypothesis:** Assume that $T(k) \leq ck - b$ for all $k < n$. Then

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1. \\ &= cn - 2b + 1 \leq cn - b \end{aligned}$$

if $b \geq 1$.

How large should c and b be? This depends on the “base” case of the recurrence.

In **Maxr**, $T(1) = 0$. Hence, we can choose $c = 1$ and $b = 1$, since it satisfies the base case. Thus, we have shown $T(n) \leq n - 1$ for all n .

Another Example

Example 2.3. Solve the following recurrence:

$$T(n) = 2T(n/2) + n.$$

Guess and verify that $T(n) \leq cn \log n$, for some constant c . **Induction hypothesis:** Assume that $T(k) \leq ck \log k$ for all $k < n$.

Hence we have:

$$\begin{aligned} T(n) &\leq 2cn/2 \log n/2 + n. \\ &= cn \log n - cn \log 2 + n. \\ &= cn \log n - cn + n. \\ &\leq cn \log n \end{aligned}$$

if $c \geq 1$. Note that c should be chosen large enough to satisfy the base condition.

A Wrong Argument using Induction If you try to show $T(n) \leq cn$ by arguing:

$$T(n) \leq 2cn/2 + n = (c+1)n = O(n).$$

This is incorrect! (why?)

Solving Recurrences via Change of Variables

Example 2.4. Solve

$$T(n) = 2T(\sqrt{n}) + 1.$$

Renaming $m = \log n$, we have:

$$T(2^m) = 2T(2^{m/2}) + 1.$$

Again, renaming $S(m) = T(2^m)$:

$$S(m) = 2S(m/2) + 1.$$

Thus, $S(m) = O(m)$ and $T(n) = T(2^m) = S(m) = O(m) = O(\log n)$.

2.8.2 A General Theorem for “Divide and Conquer” Recurrences

We now give a formula that can quickly solve certain types of recurrences. This type occurs most commonly in analysis of divide and conquer type algorithms and thus very useful.

Consider recurrences of the form:

$$T(n) = aT(n/b) + f(n),$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is some function. Assume, without loss of generality, that n is a power of b . (It can be shown that the asymptotic bounds as shown in the DC Recurrence Theorem below will hold, even when n is not a power of b . Can you show it?) Assume that $T(1) = O(1)$, i.e., the time for solving an instance of size 1 is a constant.

Theorem 2.2 (DC Recurrence Theorem). *The solution for the above recurrence $T(n)$ is:*

1. If $af(n/b) = cf(n)$ for some constant $c < 1$ then $T(n) = \Theta(f(n))$.
2. If $af(n/b) = cf(n)$ for some constant $c > 1$ then $T(n) = \Theta(n^{\log_b a})$.
3. If $af(n/b) = f(n)$ then $T(n) = \Theta(f(n) \log_b n)$.

Proof. Draw a “recursion tree” for $T(n)$: $f(n)$ is the root and it has a children each of which is a recursion tree for $T(n/b)$. That is, a recursion tree is a complete a -ary tree where each node at depth i has the value $a^i f(n/b^i)$. The leaves of the tree contains the “base cases” of the recursion. Since we are looking at asymptotic bounds, we can assume without loss of generality (w.l.o.g) that $T(1) = f(1)$. Assuming each level of the tree is full, we have,

$$T(n) = f(n) + af(n/b) + a^2 f(n/b^2) + \cdots + a^L f(n/b^L)$$

where L is the depth of the recursion tree.

Substituting, $af(n/b) = cf(n)$ in the above formula for $T(n)$, we have

$$T(n) = f(n) + cf(n) + c^2 f(n) + \cdots + c^L f(n).$$

Note that $L = \log_b n$ and since $f(1) = \Theta(1)$, $a^L f(n/b^L) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$

1) If $f(n)$ is a constant factor larger than $af(n/b)$ then $T(n)$ is a geometric series (see below) with largest term $f(n)$. Hence $T(n) = \Theta(f(n))$.

2) If $f(n)$ is a constant factor smaller than $af(n/b)$ then $T(n)$ is a geometric series with largest term $a^L f(n/b^L) = \Theta(n^{\log_b a})$.

3) If $af(n/b) = f(n)$ then there are $L + 1$ levels each level summing to $f(n)$ and hence $\Theta(f(n) \log_b n)$. $\square$

Exercise 2.3. Show the asymptotic bounds in the DC recurrence theorem hold even when n is not a power of b .

Geometric Series For real $x \neq 1$, the summation

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \cdots + x^n$$

is a geometric series and has the value

$$\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1}.$$

Thus: $\sum_{k=0}^n x^k = \Theta(x^n)$ if $x > 1$ (increasing series).

$\sum_{k=0}^n x^k = \Theta(1)$ if $x < 1$ (decreasing series).

If the series is infinite and $|x| < 1$, then

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}.$$

Example 2.5. 1. $T(n) = T(3n/4) + 2n$. Here $af(n/b) = 2(3n/4) = (3/4)f(n)$. Hence $T(n) = \Theta(n)$.

2. $T(n) = 7T(n/2) + \Theta(n^2)$. That is, $T(n) = 7T(n/2) + c_1n^2$, for some positive constant c_1 . $af(n/b) = 7c_1(n/2)^2 = (7/4)c_1n^2 = (7/4)f(n)$. Hence, $T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$.

3. MergeSort: $T(n) = 2T(n/2) + n$. Here $af(n/b) = f(n)$ and hence $T(n) = \Theta(n \log n)$.

Other Examples Some recurrences do not exactly fit the Master Theorem.

Example 2.6. $T(n) = 2T(n/2) + n/\log n$

We can't apply the Master Theorem directly since $af(n/b) = n/(\log n - 1)$ is not equal to a constant factor times $n/\log n$. We compute the recurrence directly.

The sum of the nodes in the i th level is $n/(\log n - i)$. Thus, the depth of recursion is at most $\log n - 1$.

$$T(n) = \sum_{i=0}^{\log n - 1} n/(\log n - i) = \sum_{j=1}^{\log n} n/j = nH_{\log n} = \Theta(n \log \log n),$$

where $H_k = \sum_{i=1}^k 1/i$ is the *harmonic function* and $H_k = \Theta(\log k)$.

Example 2.7. $T(n) = T(3n/4) + T(n/4) + n$.

We can solve this by directly looking at the recursion tree for this recurrence. Each complete level in the tree adds up to n and each leaf has depth between $\log_4 n$ and $\log_{4/3} n$. Hence, $T(n) = \Theta(n \log n)$.

2.8.3 Handling Floors and Ceilings in Recurrences

Suppose we have the following recurrence (this arises in MergeSort, as we will see later)

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n.$$

We want to show that $T(n) = \Theta(n \log n)$.

We show the upper bound, lower bound is similar.

$$T(n) \leq 2T(\lceil n/2 \rceil) + n \leq 2T(n/2 + 1) + n.$$

To solve for $T(n)$ we do a “domain transformation” as follows. Let $S(n) = T(n + \alpha)$ where α is a constant chosen such that

$$S(n) \leq 2S(n/2) + \Theta(n).$$

To find α , we compare the two recurrences — that of $T(n)$ and $S(n)$.

$$T(n + \alpha) \leq 2T((n + \alpha)/2 + 1) + n.$$

$$T(n + \alpha) \leq 2T(n/2 + \alpha) + \Theta(n).$$

which gives $\alpha = 2$.

By Master Theorem, $S(n) = O(n \log n)$, and hence $T(n) = S(n - \alpha) = O(n \log n)$.

2.9 Exercises

Exercise 2.4. *There are n coins placed in a row (see Figure 2.4 for an example). The goal is to form $n/2$ pairs of them by a sequence of moves. On each move a single coin can jump left or right over two coins adjacent to it (either over two single coins or one previously formed pair) to land on the next single coin — no triples are allowed. Note that the coin cannot land on an empty space (it has to land on exactly one coin). Empty space between adjacent coins is ignored.*

Find out all values of n for which the puzzle has a solution and devise an algorithm (enough to explain your algorithm in English) that solves the puzzle in the minimum number of moves for such n .

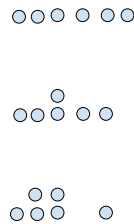


Figure 2.4: Figure for Problem 2.4. Note that there is no solution for $n = 6$.

Exercise 2.5. *Rank the following functions by order of growth; that is, find an arrangement $g_1, g_2, \dots, g_8$ of the functions satisfying $g_1 = O(g_2)$,*

$g_2 = O(g_3), \dots, g_7 = O(g_8)$. Justify your ordering.

$$(\log n)^3, n^{\log \log n}, n^{1+1/\log n}, 4^{\log n}, n!, 2^{\sqrt{2 \log n}}, n^2, (1.1)^n$$

(Hints: (1) When comparing functions of the form f^g it might be helpful to compare the logarithm of those functions, i.e., $\log f^g = g \log f$.

$n! = n(n-1) \dots (2)(1)$ which you can use to lower or upper bound $n!$, e.g., $2^n < n! < n^n$, for all $n \geq 4$. (2) Use the limit definition of the Big-O notation. You might want to recall the L'Hopital's rule for computing the limit of the ratio of two functions.))

Exercise 2.6. Prove or disprove each of the following statements:

- i. $f(n) = \Theta(g(n))$ implies $n^{f(n)} = \Theta(n^{g(n)})$
- ii. $f(n) = \Omega(f(n/2))$

Exercise 2.7. You are given a **sorted** array $S[1, \dots, n]$ of n (distinct) numbers, i.e., the numbers in the array are arranged in **increasing** order: $S[1] < S[2] < \dots < S[n]$.

Give a recursive algorithm to **search** for a given element (say x) in the array. If x is present in the array, the algorithm should return the index of the position where it is present, otherwise it should return 0.

Your algorithm should take $O(\log n)$ comparisons. Give pseudocode of your algorithm and argue that your algorithm indeed takes $O(\log n)$ comparisons (in the worst case).

Exercise 2.8. A light bulb is controlled by n switches. Each switch can be in one of two possible states — “OFF” and “ON”. We can't tell the status of a switch (i.e., whether it is ON or OFF) at any given time. We can only toggle it between the two states by pressing the switch. Only when **all** switches are in the ON state, the light bulb will glow, otherwise it will not glow. Given that the light bulb is not initially glowing, give a **recursive** algorithm to make it glow. What is the number of operations (as a function of n) needed by your algorithm in the worst case? (Count each toggle of a switch as one operation.)

Exercise 2.9. Consider the problem of finding the largest and second largest elements from a set of n elements.

- a) Give a simple iterative algorithm that takes no more than $2n - 3$ comparisons.
- b) Give a divide and conquer algorithm and show that it takes $\frac{3n}{2} - 2$ comparisons when n is a power of 2.

- c) Show how you can find both these elements in $n + \lceil \log n \rceil - 2$ comparisons in the worst case. (Hint: See hint in problem 2.1.)

Exercise 2.10. Solve the following recurrences. Give the answer in terms of **Big-Theta** notation. Solve up to constant factors, i.e., your answer must give the correct function for $T(n)$, up to constant factors. You can assume constant base cases, i.e., $T(1) = T(0) = c$, where c is a positive constant. You can ignore floors and ceilings. You can use the DC recurrence theorem if it applies.

- a) $T(n) = 3T(n/2) + n$
b) $T(n) = 4T(n/2) + n \log n$
c) $T(n) = \sqrt{n}T(\sqrt{n}) + n$
d) $T(n) = T(n/2) + T(n/3) + 1.$

Exercise 2.11. You are given an array consisting of n numbers. A popular element is an element that occurs (strictly) **more than** $n/2$ times in the array. Give an algorithm that finds the popular element in the array if it exists, otherwise it should output “NO”. Your algorithm should take no more than $2n$ comparisons. (As usual, we only count the comparisons between array elements.) Give pseudocode and show that your algorithm indeed takes no more than $2n$ comparisons. (Hint: See the hint for Problem 2.1.)