MASSIMO DI PIERRO

ANNOTATED ALGORITHMS IN PYTHON

WITH APPLICATIONS IN PHYSICS, BIOLOGY, AND FINANCE

EXPERTS4SOLUTIONS

For more information about appropriate use of this material, contact:

Massimo Di Pierro
School of Computing
DePaul University
243 S Wabash Ave
Chicago, IL 60604 (USA)
Email: massimo.dipierro@gmail.com

Library of Congress Cataloging-in-Publication Data:

ISBN: 978-0-9911604-0-2

Build Date: December 4, 2013

to my parents

Contents

3

Theory of Algorithms

An algorithm is a step-by-step procedure for solving a problem and is typically developed before doing any programming. The word comes from *algorism*, from the mathematician al-Khwarizmi, and was used to refer to the rules of performing arithmetic using Hindu–Arabic numerals and the systematic solution of equations.

In fact, algorithms are independent of any programming language. Efficient algorithms can have a dramatic effect on our problem-solving capabilities.

The basic steps of algorithms are loops (`for`, conditionals (`if`), and function calls. Algorithms also make use of arithmetic expressions, logical expressions (`not`, `and`, `or`), and expressions that can be reduced to the other basic components.

The issues that concern us when developing and analyzing algorithms are the following:

1. Correctness: of the problem specification, of the proposed algorithm, and of its implementation in some programming language (we will not worry about the third one; program verification is another subject altogether)

2. Amount of work done: for example, running time of the algorithm in terms of the input size (independent of hardware and programming

language)

3. Amount of space used: here we mean the amount of extra space (system resources) beyond the size of the input (independent of hardware and programming language); we will say that an algorithm is *in place* if the amount of extra space is constant with respect to input size

4. Simplicity, clarity: unfortunately, the simplest is not always the best in other ways

5. Optimality: can we prove that it does as well as or better than any other algorithm?

3.1 Order of growth of algorithms

The *insertion sort* is a simple algorithm in which an array of elements is sorted in place, one entry at a time. It is not the fastest sorting algorithm, but it is simple and does not require extra memory other than the memory needed to store the input array.

The insertion sort works by iterating. Every iteration i of the insertion sort removes one element from the input data and inserts it into the correct position in the already-sorted subarray `A[j]` for $0 \leq j < i$. The algorithm iterates n times (where n is the total size of the input array) until no input elements remain to be sorted:

```
def insertion_sort(A):
    for i in xrange(1,len(A)):
        for j in xrange(i,0,-1):
            if A[j]<A[j-1]:
                A[j], A[j-1] = A[j-1], A[j]
            else: break
```

Here is an example:

```
>>> import random
>>> a=[random.randint(0,100) for k in xrange(20)]
>>> insertion_sort(a)
>>> print(a)
[6, 8, 9, 17, 30, 31, 45, 48, 49, 56, 56, 57, 65, 66, 75, 75, 82, 89, 90, 99]
```

One important question is, how long does this algorithm take to run?

How does its running time scale with the input size?

Given any algorithm, we can define three characteristic functions:

- $T_{worst}(n)$: the running time in the worst case
- $T_{best}(n)$: the running time in the best case
- $T_{average}(n)$: the running time in the average case

The best case for an insertion sort is realized when the input is already sorted. In this case, the inner for loop exits (breaks) always at the first iteration, thus only the most outer loop is important, and this is proportional to n; therefore $T_{best}(n) \propto n$. The worst case for the insertion sort is realized when the input is sorted in reversed order. In this case, we can prove, and we do so subsequently, that $T_{worst}(n) \propto n^2$. For this algorithm, a statistical analysis shows that the worst case is also the average case.

Often we cannot determine exactly the running time function, but we may be able to set bounds to the running time.

We define the following sets:

- $O(g(n))$: the set of functions that grow no faster than $g(n)$ when $n \to \infty$
- $\Omega(g(n))$: the set of functions that grow no slower than $g(n)$ when $n \to \infty$
- $\Theta(g(n))$: the set of functions that grow at the same rate as $g(n)$ when $n \to \infty$
- $o(g(n))$: the set of functions that grow slower than $g(n)$ when $n \to \infty$
- $\omega(g(n))$: the set of functions that grow faster than $g(n)$ when $n \to \infty$

We can rewrite the preceding definitions in a more formal way:

$$O(g(n)) \equiv \{f(n) : \exists n_0, c_0,\ \forall n > n_0,\ 0 \leq f(n) < c_0 g(n)\} \quad (3.1)$$
$$\Omega(g(n)) \equiv \{f(n) : \exists n_0, c_0,\ \forall n > n_0,\ 0 \leq c_0 g(n) < f(n)\} \quad (3.2)$$
$$\Theta(g(n)) \equiv O(g(n)) \cap \Omega(g(n)) \quad (3.3)$$
$$o(g(n)) \equiv O(g(n)) - \Omega(g(n)) \quad (3.4)$$
$$\omega(g(n)) \equiv \Omega(g(n)) - O(g(n)) \quad (3.5)$$

We can also provide a practical rule to determine if a function f belongs to one of the previous sets defined by g.

Compute the limit

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = a \quad (3.6)$$

and look up the result in the following table:

$$\begin{array}{lcl} a \text{ is positive or zero} & \Longrightarrow & f(n) \in O(g(n)) \Leftrightarrow f \preceq g \\ a \text{ is positive or infinity} & \Longrightarrow & f(n) \in \Omega(g(n)) \Leftrightarrow f \succeq g \\ a \text{ is positive} & \Longrightarrow & f(n) \in \Theta(g(n)) \Leftrightarrow f \sim g \\ a \text{ is zero} & \Longrightarrow & f(n) \in o(g(n)) \Leftrightarrow f \prec g \\ a \text{ is infinity} & \Longrightarrow & f(n) \in \omega(g(n)) \Leftrightarrow f \succ g \end{array} \quad (3.7)$$

Notice the preceding practical rule assumes the limits exist.

Here is an example:

Given $f(n) = n \log n + 3n$ and $g(n) = n^2$

$$\lim_{n\to\infty} \frac{n \log n + 3n}{n^2} \stackrel{l'Hopital}{\longrightarrow} \lim_{n\to\infty} \frac{1/n}{2} = 0 \quad (3.8)$$

we conclude that $n \log n + 3n$ is in $O(n^2)$.

Given an algorithm A that acts on input of size n, we say that the algorithm is $O(g(n))$ if its worst running time as a function of n is in $O(g(n))$. Similarly, we say that the algorithm is in $\Omega(g(n))$ if its best running time is in $\Omega(g(n))$. We also say that the algorithm is in $\Theta(g(n))$ if both its best running time and its worst running time are in $\Theta(g(n))$.

More formally, we can write the following:

$$T_{worst}(n) \in O(g(n)) \quad \Rightarrow \quad A \in O(g(n)) \tag{3.9}$$
$$T_{best}(n) \in \Omega(g(n)) \quad \Rightarrow \quad A \in \Omega(g(n)) \tag{3.10}$$
$$A \in O(g(n))\text{and}A \in O(g(n)) \quad \Rightarrow \quad A \in \Theta(g(n)) \tag{3.11}$$
$$\tag{3.12}$$

We still have not solved the problem of computing the best, average, and worst running times.

3.1.1 Best and worst running times

The procedure for computing the worst and best running times is similar. It is simple in theory but difficult in practice because it requires an understanding of the algorithm's inner workings.

Consider the following algorithm, which finds the minimum of an array or list A:

```python
def find_minimum(A):
    minimum = a[0]
    for element in A:
        if element < minimum:
            minimum = element
    return minimum
```

To compute the running time in the worst case, we assume that the maximum number of computations is performed. That happens when the if statements are always `True`. To compute the best running time, we assume that the minimum number of computations is performed. That happens when the if statement is always `False`. Under each of the two scenarios, we compute the running time by counting how many times the most nested operation is performed.

In the preceding algorithm, the most nested operation is the evaluation of the `if` statement, and that is executed for each element in `A`; for example, assuming `A` has n elements, the `if` statement will be executed n times.

Therefore both the best and worst running times are proportional to n, thus making this algorithm $O(n)$, $\Omega(n)$, and $\Theta(n)$.

More formally, we can observe that this algorithm performs the following operations:

- One assignment (line 2)
- Loops $n =$`len(A)` times (line 3)
- For each loop iteration, performs one comparison (line 4)
- Line 5 is executed only if the condition is true

Because there are no nested loops, the time to execute each loop iteration is about the same, and the running time is proportional to the number of loop iterations.

For a loop iteration that does not contain further loops, the time it takes to compute each iteration, its `running time`, is constant (therefore equal to 1). For algorithms that contain nested loops, we will have to evaluate nested sums.

Here is the simplest example:

```
def loop0(n):
    for i in xrange(0,n):
        print(i)
```

which we can map into

$$T(n) = \sum_{i=0}^{i<n} 1 = n \in \Theta(n) \Rightarrow \texttt{loop0} \in \Theta(n) \tag{3.13}$$

Here is a similar example where we have a single loop (corresponding to a single sum) that loops n^2 times:

```
def loop1(n):
    for i in xrange(0,n*n):
        print(i)
```

and here is the corresponding running time formula:

$$T(n) = \sum_{i=0}^{i<n^2} 1 = n^2 \in \Theta(n^2) \Rightarrow \texttt{loop1} \in \Theta(n^2) \tag{3.14}$$

The following provides an example of nested loops:

```
def loop2(n):
    for i in xrange(0,n):
        for j in xrange(0,n):
            print(i,j)
```

Here the time for the inner loop is directly determined by n and does not depend on the outer loop's counter; therefore

$$T(n) = \sum_{i=0}^{i<n} \sum_{j=0}^{j<n} 1 = \sum_{i=0}^{i<n} n = n^2 + ... \in \Theta(n^2) \Rightarrow \texttt{loop2} \in \Theta(n^2) \tag{3.15}$$

This is not always the case. In the following code, the inner loop does depend on the value of the outer loop:

```
def loop3(n):
    for i in xrange(0,n):
        for j in xrange(0,i):
            print(i,j)
```

Therefore, when we write its running time in terms of a sum, care must be taken that the upper limit of the inner sum is the upper limit of the outer sum:

$$T(n) = \sum_{i=0}^{i<n} \sum_{j=0}^{j<i} 1 = \sum_{i=0}^{i<n} i = \frac{1}{2} n(n-1) \in \Theta(n^2) \Rightarrow \texttt{loop3} \in \Theta(n^2) \tag{3.16}$$

The appendix of this book provides examples of typical sums that come up in these types of formulas and their solutions.

Here is one more example falling in the same category, although the inner loop depends quadratically on the index of the outer loop:

Example: loop4

```
def loop4(n):
    for i in xrange(0,n):
        for j in xrange(0,i*i):
            print(i,j)
```

Therefore the formula for the running time is more complicated:

$$T(n) = \sum_{i=0}^{i<n}\sum_{j=0}^{j<i^2} 1 = \sum_{i=0}^{i<n} i^2 = \frac{1}{6}n(n-1)(2n-1) \in \Theta(n^3) \quad (3.17)$$

$$\Rightarrow \texttt{loop4} \in \Theta(n^3) \quad (3.18)$$

If the algorithm does not contain nested loops, then we need to compute the running time of each loop and take the maximum:

Example: concatenate0

```
def concatenate0(n):
    for i in xrange(n*n):
        print(i)
    for j in xrange(n*n*n):
        print(j)
```

$$T(n) = \Theta(\max(n^2, n^3)) \Rightarrow \texttt{concatenate0} \in \Theta(n^3) \quad (3.19)$$

If there is an if statement, we need to compute the running time for each condition and pick the maximum when computing the worst running time, or the minimum for the best running time:

```
def concatenate1(n):
    if a<0:
        for i in xrange(n*n):
            print(i)
    else:
        for j in xrange(n*n*n):
            print(j)
```

$$T_{worst}(n) = \Theta(\max(n^2, n^3)) \Rightarrow \texttt{concatenate1} \in (n^3) \quad (3.20)$$

$$T_{best}(n) = \Theta(\min(n^2, n^3)) \Rightarrow \texttt{concatenate1} \in \Omega(n^2) \quad (3.21)$$

This can be expressed more formally as follows:

$$O(f(n)) + \Theta(g(n)) = \Theta(g(n)) \text{ iff } f(n) \in O(g(n)) \tag{3.22}$$
$$\Theta(f(n)) + \Theta(g(n)) = \Theta(g(n)) \text{ iff } f(n) \in O(g(n)) \tag{3.23}$$
$$\Omega(f(n)) + \Theta(g(n)) = \Omega(f(n)) \text{ iff } f(n) \in \Omega(g(n)) \tag{3.24}$$

which we can apply as in the following example:

$$T(n) = [\underbrace{n^2 + n + 3}_{\Theta(n^2)} + \underbrace{e^n - \log n}_{\Theta(e^n)}] \in \Theta(e^n) \text{ because } n^2 \in O(e^n) \tag{3.25}$$

3.2 Recurrence relations

The *merge sort* [13] is another sorting algorithm. It is faster than the insertion sort. It was invented by John von Neumann, the physicist credited for inventing also modern computer architecture and game theory.

The merge sort works as follows.

If the input array has length 0 or 1, then it is already sorted, and the algorithm does not perform any other operation.

If the input array has a length greater than 1, it divides the array into two subsets of about half the size. Each subarray is sorted by applying the merge sort recursively (it calls itself!). It then merges the two subarrays back into one sorted array (this step is called *merge*).

Consider the following Python implementation of the merge sort:

```
def mergesort(A, p=0, r=None):
    if r is None: r = len(A)
    if p<r-1:
        q = int((p+r)/2)
        mergesort(A,p,q)
        mergesort(A,q,r)
        merge(A,p,q,r)

def merge(A,p,q,r):
```

```
    B,i,j = [],p,q
    while True:
        if A[i]<=A[j]:
            B.append(A[i])
            i=i+1
        else:
            B.append(A[j])
            j=j+1
        if i==q:
            while j<r:
                B.append(A[j])
                j=j+1
            break
        if j==r:
            while i<q:
                B.append(A[i])
                i=i+1
            break
    A[p:r]=B
```

Because this algorithm calls itself *recursively*, it is more difficult to compute its running time.

Consider the `merge` function first. At each step, it increases either i or j, where i is always in between p and q and j is always in between q and r. This means that the running time of the merge is proportional to the total number of values they can span from p to r. This implies that

$$\text{merge} \in \Theta(r-p) \tag{3.26}$$

We cannot compute the running time of the `mergesort` function using the same direct analysis, but we can assume its running time is $T(n)$, where $n = r - p$ and n is the size of the input data to be sorted and also the difference between its two arguments p and r. We can express this running time in terms of its components:

- It calls itself twice on half of the input data, $2T(n/2)$
- It calls the merge once on the entire data, $\Theta(n)$

We can summarize this into

$$T(n) = 2T(n/2) + n \tag{3.27}$$

This is called a *recurrence relation*. We turned the problem of computing the running time of the algorithm into the problem of solving the recurrence relation. This is now a math problem.

Some recurrence relations can be difficult to solve, but most of them follow in one of these categories:

$$T(n) = aT(n-b) + \Theta(f(n)) \Rightarrow T(n) \in \Theta(max(a^n, nf(n))) \tag{3.28}$$
$$T(n) = T(b) + T(n-b-a) + \Theta(f(n)) \Rightarrow T(n) \in \Theta(nf(n)) \tag{3.29}$$
$$T(n) = aT(n/b) + \Theta(n^m) \text{ and } a < b^m \Rightarrow T(n) \in \Theta(n^m) \tag{3.30}$$
$$T(n) = aT(n/b) + \Theta(n^m) \text{ and } a = b^m \Rightarrow T(n) \in \Theta(n^m \log n) \tag{3.31}$$
$$T(n) = aT(n/b) + \Theta(n^m) \text{ and } a > b^m \Rightarrow T(n) \in \Theta(n^{\log_b a}) \tag{3.32}$$
$$T(n) = aT(n/b) + \Theta(n^m \log^p n) \text{ and } a < b^m \Rightarrow T(n) \in \Theta(n^m \log^p n) \tag{3.33}$$
$$T(n) = aT(n/b) + \Theta(n^m \log^p n) \text{ and } a = b^m \Rightarrow T(n) \in \Theta(n^m \log^{p+1} n) \tag{3.34}$$
$$T(n) = aT(n/b) + \Theta(n^m \log^p n) \text{ and } a > b^m \Rightarrow T(n) \in \Theta(n^{\log_b a}) \tag{3.35}$$
$$T(n) = aT(n/b) + \Theta(q^n) \Rightarrow T(n) \in \Theta(q^n) \tag{3.36}$$
$$T(n) = aT(n/a-b) + \Theta(f(n)) \Rightarrow T(n) \in \Theta(f(n)\log(n)) \tag{3.37}$$

(they work for $m \geq 0$, $p \geq 0$, and $q > 1$).

These results are a practical simplification of a theorem known as the *master theorem* [14].

3.2.1 Reducible recurrence relations

Other recurrence relations do not immediately fit one of the preceding patterns, but often they can be reduced (transformed) to fit.

Consider the following recurrence relation:

$$T(n) = 2T(\sqrt{n}) + \log n \tag{3.38}$$

We can replace n with $e^k = n$ in eq. (3.38) and obtain

$$T(e^k) = 2T(e^{k/2}) + k \tag{3.39}$$

If we also replace $T(e^k)$ with $S(k) = T(e^k)$, we obtain

$$\underbrace{S(k)}_{T(e^k)} = 2\,\underbrace{S(k/2)}_{T(e^{k/2})} + k \tag{3.40}$$

so that we can now apply the master theorem to S. We obtain that $S(k) \in \Theta(k \log k)$. Once we have the order of growth of S, we can determine the order of growth of $T(n)$ by substitution:

$$T(n) = S(\log n) \in \Theta(\underbrace{\log n}_{k} \log \underbrace{\log n}_{k}) \tag{3.41}$$

Note that there are recurrence relations that cannot be solved with any of the methods described.

Here are some examples of recursive algorithms and their corresponding recurrence relations with solution:

```
def factorial1(n):
    if n==0:
        return 1
    else:
        return n*factorial1(n-1)
```

$$T(n) = T(n-1) + 1 \Rightarrow T(n) \in \Theta(n) \Rightarrow \texttt{factorial1} \in \Theta(n) \tag{3.42}$$

```
def recursive0(n):
    if n==0:
        return 1
    else:
        loop3(n)
        return n*n*recursive0(n-1)
```

$$T(n) = T(n-1) + P_2(n) \Rightarrow T(n) \in \Theta(n^2) \Rightarrow \texttt{recursive0} \in \Theta(n^3) \tag{3.43}$$

```
def recursive1(n):
    if n==0:
        return 1
    else:
        loop3(n)
        return n*recursive1(n-1)*recursive1(n-1)
```

$$T(n) = 2T(n-1) + P_2(n) \Rightarrow T(n) \in \Theta(2^n) \Rightarrow \texttt{recursive1} \in \Theta(2^n) \tag{3.44}$$

```
def recursive2(n):
    if n==0:
        return 1
    else:
        a=factorial0(n)
        return a*recursive2(n/2)*recursive1(n/2)
```

$$T(n) = 2T(n/2) + P_1(n) \Rightarrow T(n) \in \Theta(n \log n) \Rightarrow \texttt{recursive2} \in \Theta(n \log n) \tag{3.45}$$

One example of practical interest for us is the binary search below. It finds the location of the element in a sorted input array A:

```
def binary_search(A,element):
    a,b = 0, len(A)-1
    while b>=a:
        x = int((a+b)/2)
        if A[x]<element:
            a = x+1
        elif A[x]>element:
            b = x-1
        else:
            return x
    return None
```

Notice that this algorithm does not appear to be recursive, but in practice, it is because of the apparently infinite while loop. The content of the while loop runs in constant time and then loops again on a problem of half of the original size:

$$T(n) = T(n/2) + 1 \Rightarrow \texttt{binary\_search} \in \Theta(\log n) \tag{3.46}$$

The idea of the `binary_search` is used in the bisection method for solving nonlinear equations.

Do not confuse `T` notation with Θ notation:

The theta notation can also be used to describe the memory used by an

Algorithm	Recurrence Relationship	Running time
Binary Search	$T(n) = T(\frac{n}{2}) + \Theta(1)$	$\Theta(log(n))$
Binary Tree Traversal	$T(n) = 2T(\frac{n}{2}) + \Theta(1)$	$\Theta(n)$
Optimal Sorted Matrix Search	$T(n) = 2T(\frac{n}{2}) + \Theta(log(n))$	$\Theta(n)$
Merge Sort	$T(n) = T(\frac{n}{2}) + \Theta(n)$	$\Theta(nlog(n))$

algorithm as a function of the input, T_{memory}, as well as its running time.

3.3 Types of algorithms

Divide-and-conquer is a method of designing algorithms that (informally) proceeds as follows: given an instance of the problem to be solved, split this into several, smaller sub-instances (of the same problem), independently solve each of the sub-instances and then combine the sub-instance solutions to yield a solution for the original instance. This description raises the question, by what methods are the sub-instances to be independently solved? The answer to this question is central to the concept of the divide-and-conquer algorithm and is a key factor in gauging their efficiency. The solution is unique for each problem.

The merge sort algorithm of the previous section is an example of a divide-and-conquer algorithm. In the merge sort, we sort an array by dividing it into two arrays and recursively sorting (conquering) each of the smaller arrays.

Most divide-and-conquer algorithms are recursive, although this is not a requirement.

Dynamic programming is a paradigm that is most often applied in the construction of algorithms to solve a certain class of optimization problems, that is, problems that require the minimization or maximization of some measure. One disadvantage of using divide-and-conquer is that the process of recursively solving separate sub-instances can result in the same computations being performed repeatedly because identical sub-instances may arise. For example, if you are computing the path between two nodes in a graph, some portions of multiple paths will follow the same last few hops. Why compute the last few hops for every path when you would get the same result every time?

The idea behind dynamic programming is to avoid this pathology by obviating the requirement to calculate the same quantity twice. The method usually accomplishes this by maintaining a table of sub-instance results. We say that dynamic programming is a bottom-up technique in which the smallest sub-instances are explicitly solved first and the results of these are used to construct solutions to progressively larger sub-instances. In contrast, we say that the divide-and-conquer is a top-down technique.

We can refactor the `mergesort` algorithm to eliminate recursion in the algorithm implementation, while keeping the logic of the algorithm unchanged. Here is a possible implementation:

```
def mergesort_nonrecursive(A):
    blocksize, n = 1, len(A)
    while blocksize<n:
        for p in xrange(0, n, 2*blocksize):
            q = p+blocksize
            r = min(q+blocksize, n)
            if r>q:
                Merge(A,p,q,r)
        blocksize = 2*blocksize
```

Notice that this has the same running time as the original `mergesort` because, although it is not recursive, it performs the same operations:

$$T_{best} \in \Theta(n \log n) \tag{3.47}$$
$$T_{average} \in \Theta(n \log n) \tag{3.48}$$
$$T_{worst} \in \Theta(n \log n) \tag{3.49}$$
$$T_{memory} \in \Theta(1) \tag{3.50}$$

Greedy algorithms work in phases. In each phase, a decision is made that appears to be good, without regard for future consequences. Generally, this means that some local optimum is chosen. This "take what you can get now" strategy is the source of the name for this class of algorithms. When the algorithm terminates, we hope that the local optimum is equal to the global optimum. If this is the case, then the algorithm is correct; otherwise, the algorithm has produced a suboptimal solution. If the best answer is not required, then simple greedy algorithms are sometimes used to generate approximate answers, rather than using the more complicated algorithms generally required to generate an exact answer. Even for problems that can be solved exactly by a greedy algorithm, establishing the correctness of the method may be a nontrivial process.

For example, computing change for a purchase in a store is a good case of a greedy algorithm. Assume you need to give change back for a purchase. You would have three choices:

- Give the smallest denomination repeatedly until the correct amount is returned
- Give a random denomination repeatedly until you reach the correct amount. If a random choice exceeds the total, then pick another denomination until the correct amount is returned
- Give the largest denomination less than the amount to return repeatedly until the correct amount is returned

In this case, the third choice is the correct one.

Other types of algorithms do not fit into any of the preceding categories. One is, for example, backtracking. Backtracking is not covered in this course.

3.3.1 Memoization

One case of a top-down approach that is very general and falls under the umbrella of dynamic programming is called *memoization*. Memoization consists of allowing users to write algorithms using a naive divide-and-conquer approach, but functions that may be called more than once are modified so that their output is cached, and if they are called again with the same initial state, instead of the algorithm running again, the output is retrieved from the cache and returned without any computations.

Consider, for example, Fibonacci numbers:

$$\mathrm{Fib}(0) = 0 \tag{3.51}$$
$$\mathrm{Fib}(1) = 1 \tag{3.52}$$
$$\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) \text{ for } n > 1 \tag{3.53}$$

which we can implement using divide-and-conquer as follows:

```
def fib(n):
    return n if n<2 else fib(n-1)+fib(n-2)
```

The recurrence relation for this algorithm is $T(n) = T(n-1) + T(n-2) + 1$, and its solution can be proven to be exponential. This is because this algorithm calls itself more than necessary with the same input values and keeps solving the same subproblem over and over.

Python can implement memoization using the following decorator:

Listing 3.1: in file: `nlib.py`

```
class memoize(object):
    def __init__ (self, f):
        self.f = f
        self.storage = {}
    def __call__ (self, *args, **kwargs):
        key = str((self.f.__name__, args, kwargs))
        try:
            value = self.storage[key]
        except KeyError:
            value = self.f(*args, **kwargs)
            self.storage[key] = value
        return value
```

and simply decorating the recursive function as follows:

Listing 3.2: in file: `nlib.py`

```
@memoize
def fib(n):
    return n if n<2 else fib(n-1)+fib(n-2)
```

which we can call as

Listing 3.3: in file: `nlib.py`

```
>>> print(fib(11))
89
```

A decorator is a Python function that takes a function and returns a callable object (or a function) to replace the one passed as input. In the previous example, we are using the `@memoize` decorator to replace the `fib` function with the `__call__` argument of the `memoize` class.

This makes the algorithm run much faster. Its running time goes from exponential to linear. Notice that the preceding `memoize` decorator is very general and can be used to decorate any other function.

One more direct dynamic programming approach consists in removing the recursion:

```
def fib(n):
    if n < 2: return n
    a, b = 0, 1
    for i in xrange(1,n):
        a, b = b, a+b
    return b
```

This also makes the algorithm linear and $T(n) \in \Theta(n)$.

Notice that we easily modify the memoization algorithm to store the partial results in a shared space, for example, on disk using the `PersistentDictionary`:

Listing 3.4: in file: `nlib.py`

```
class memoize_persistent(object):
    STORAGE = 'memoize.sqlite'
    def __init__ (self, f):
        self.f = f
        self.storage = PersistentDictionary(memoize_persistent.STORAGE)
    def __call__ (self, *args, **kwargs):
        key = str((self.f.__name__, args, kwargs))
        try:
            value = self.storage[key]
        except KeyError:
            value = self.f(*args, **kwargs)
            self.storage[key] = value
        return value
```

We can use it as we did before, but we can now start and stop the program or run concurrent parallel programs, and as long as they have access to the “memoize.sqlite” file, they will share the cache.

3.4 Timing algorithms

The order of growth is a theoretical concept. In practice, we need to time algorithms to check if findings are correct and, more important, to determine the magnitude of the constants in the T functions.

For example, consider this:

```
def f1(n):
    return sum(g1(x) for x in range(n))

def f2(n):
    return sum(g2(x) for x in range(n**2))
```

Since `f1` is $\Theta(n)$ and `f2` is $\Theta(n^2)$, we may be led to conclude that the latter is slower. It may very well be that `g1` is 10^6 smaller than `g2` and therefore $T_{f1}(n) = c_1 n$, $T_{f2}(n) = c_2 n^2$, but if $c_1 = 10^6 c_2$, then $T_{f1}(n) > T_{f2}(n)$ when $n < 10^6$.

To time functions in Python, we can use this simple algorithm:

```
def timef(f, ns=1000, dt = 60):
    import time
    t = t0 = time.time()
    for k in xrange(1,ns):
        f()
        t = time.time()
        if t-t0>dt: break
    return (t-t0)/k
```

This function calls and averages the running time of `f()` for the minimum between `ns=1000` iterations and `dt=60` seconds.

It is now easy, for example, to time the fib function without memoize,

```
>>> def fib(n):
...     return n if n<2 else fib(n-1)+fib(n-2)
>>> for k in range(15,20):
...     print k,timef(lambda:fib(k))
15 0.000315684575338
16 0.000576375363706
17 0.000936052104732
18 0.00135168084153
19 0.00217730337912
```

and with memoize,

```
>>> @memoize
```

```
... def fib(n):
...     return n if n<2 else fib(n-1)+fib(n-2)
>>> for k in range(15,20):
...     print k,timef(lambda:fib(k))
15 4.24022311802e-06
16 4.02901146386e-06
17 4.21922128122e-06
18 4.02495429084e-06
19 3.73784963552e-06
```

The former shows an exponential behavior; the latter does not.

3.5 Data structures

3.5.1 Arrays

An array is a data structure in which a series of numbers are stored contiguously in memory. The time to access each number (to read or write it) is constant. The time to remove, append, or insert an element may require moving the entire array to a more spacious memory location, and therefore, in the worst case, the time is proportional to the size of the array.

Arrays are the appropriate containers when the number of elements does not change often and when elements have to be accessed in random order.

3.5.2 List

A list is a data structure in which data are not stored contiguously, and each element has knowledge of the location of the next element (and perhaps of the previous element, in a doubly linked list). This means that accessing any element for (read and write) requires finding the element and therefore looping. In the worst case, the time to find an element is proportional to the size of the list. Once an element has been found, any operation on the element, including read, write, delete, and insert, before or after can be done in constant time.

Lists are the appropriate choice when the number of elements can vary often and when their elements are usually accessed sequentially via iterations.

In Python, what is called a `list` is actually an array of pointers to the elements.

3.5.3 Stack

A stack data structure is a container, and it is usually implemented as a list. It has the property that the first thing you can take out is the last thing put in. This is commonly known as last-in, first-out, or LIFO. The method to insert or add data to the container is called *push*, and the method to extract data is called *pop*.

In Python, we can implement push by appending an item at the end of a list (Python already has a method for this called `.append`), and we can implement pop by removing the last element of a list and returning it (Python has a method for this called `.pop`).

A simple stack example is as follows:

```
>>> stk = []
>>> stk.append("One")
>>> stk.append("Two")
>>> print stk.pop()
Two
>>> stk.append("Three")
>>> print stk.pop()
Three
>>> print stk.pop()
One
```

3.5.4 Queue

A queue data structure is similar to a stack but, whereas the stack returns the most recent item added, a queue returns the oldest item in the list. This is commonly called first-in, first-out, or FIFO. To use Python lists to implement a queue, insert the element to add in the first position of the list as follows:

```
>>> que = []
>>> que.insert(0,"One")
>>> que.insert(0,"Two")
>>> print que.pop()
One
>>> que.insert(0,"Three")
>>> print que.pop()
Two
>>> print que.pop()
Three
```

Lists in Python are not an efficient mechanism for implementing queues. Each insertion or removal of an element at the front of a list requires all the elements in the list to be shifted by one. The Python package `collections.deque` is designed to implement queues and stacks. For a stack or queue, you use the same method `.append` to add items. For a stack, `.pop` is used to return the most recent item added, while to build a queue, use `.popleft` to remove the oldest item in the list:

```
>>> from collections import deque
>>> que = deque([])
>>> que.append("One")
>>> que.append("Two")
>>> print que.popleft()
One
>>> que.append("Three")
>>> print que.popleft()
Two
>>> print que.popleft()
Three
```

3.5.5 Sorting

In the previous sections, we have seen the *insertion sort* and the *merge sort*. Here we consider, as examples, other sorting algorithms: the *quicksort* [13], the *randomized quicksort*, and the *counting sort*:

```
def quicksort(A,p=0,r=-1):
    if r is -1:
        r=len(A)
    if p<r-1:
        q=partition(A,p,r)
        quicksort(A,p,q)
        quicksort(A,q+1,r)

```

```
def partition(A,i,j):
    x=A[i]
    h=i
    for k in xrange(i+1,j):
        if A[k]<x:
            h=h+1
            A[h],A[k] = A[k],A[h]
    A[h],A[i] = A[i],A[h]
    return h
```

The running time of the quicksort is given by

$$T_{best} \in \Theta(n \log n) \tag{3.54}$$
$$T_{average} \in \Theta(n \log n) \tag{3.55}$$
$$T_{worst} \in \Theta(n^2) \tag{3.56}$$
$$\tag{3.57}$$

The quicksort can also be randomized by picking the pivot, `A[r]`, at random:

```
def quicksort(A,p=0,r=-1):
    if r is -1:
        r=len(A)
    if p<r-1:
        q = random.randint(p,r-1)
        A[p], A[q] = A[q], A[p]
        q=partition(A,p,r)
        quicksort(A,p,q)
        quicksort(A,q+1,r)
```

In this case, the best and the worst running times do not change, but the average improves when the input is already almost sorted.

The *counting sort* algorithm is special because it only works for arrays of positive integers. This extra requirement allows it to run faster than other sorting algorithms, under some conditions. In fact, this algorithm is linear in the range span by the elements of the input array.

Here is a possible implementation:

```
def countingsort(A):
    if min(A)<0:
        raise '_counting_sort List Unbound'
```

```
    i, n, k = 0, len(A), max(A)+1
    C = [0]*k
    for j in xrange(n):
        C[A[j]] = C[A[j]]+1
    for j in xrange(k):
        while C[j]>0:
           (A[i], C[j], i) = (j, C[j]-1, i+1)
```

If we define $k = max(A) - min(A) + 1$ and $n = len(A)$, we see

$$T_{best} \in \Theta(k+n) \tag{3.58}$$
$$T_{average} \in \Theta(k+n) \tag{3.59}$$
$$T_{worst} \in \Theta(k+n) \tag{3.60}$$
$$T_{memory} \in \Theta(k) \tag{3.61}$$

Notice that here we have also computed T_{memory}, for example, the order of growth of memory (not of time) as a function of the input size. In fact, this algorithm differs from the previous ones because it requires a temporary array C.

3.6 Tree algorithms

3.6.1 Heapsort and priority queues

Consider a *complete binary tree* as the one in the following figure:

It starts from the top node, called the *root*. Each node has zero, one, or two children. It is called complete because nodes have been added from top to bottom and left to right, filling available slots. We can think of each level of the tree as a generation, where the older generation consists of one node, the next generation of two, the next of four, and so on. We can also number nodes from top to bottom and left to right, as in the image. This allows us to map the elements of a complete binary tree into the elements of an array.

We can implement a complete binary tree using a list, and the child–parent relations are given by the following formulas:

```
def heap_parent(i):
```

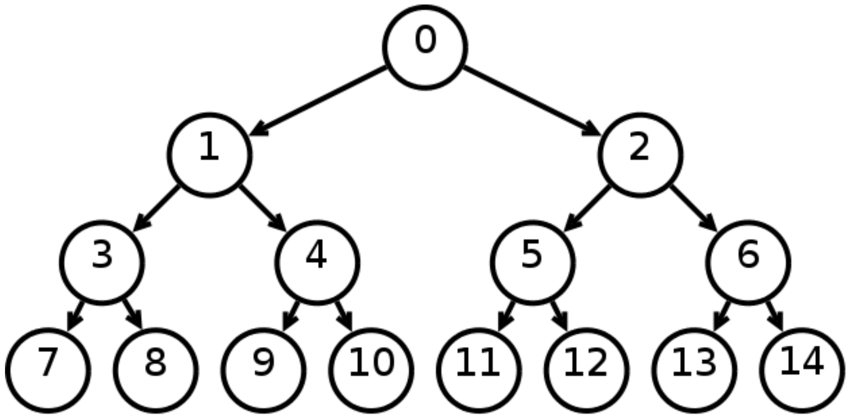

Figure 3.1: Example of a heap data structure. The number represents not the data in the heap but the numbering of the nodes.

```
    return int((i-1)/2)

def heap_left_child(i):
    return 2*i+1

def heap_right_child(i):
    return 2*i+2
```

We can store data (e.g., numbers) in the nodes (or in the corresponding array). If the data are stored in such a way that the value at one node is always greater or equal than the value at its children, the array is called a *heap* and also a *priority queue*.

First of all, we need an algorithm to convert a list into a heap:

```
def heapify(A):
    for i in xrange(int(len(A)/2)-1,-1,-1):
        heapify_one(A,i)

def heapify_one(A,i,heapsize=None):
    if heapsize is None:
        heapsize = len(A)
    left = 2*i+1
    right = 2*i+2
    if left<heapsize and A[left]>A[i]:
        largest = left
    else:
        largest = i
    if right<heapsize and A[right]>A[largest]:
        largest = right
```

```
    if largest!=i:
        (A[i], A[largest]) = (A[largest], A[i])
        heapify_one(A,largest,heapsize)
```

Now we can call `build_heap` on any array or list and turn it into a heap. Because the first element is by definition the smallest, we can use the heap to sort numbers in three steps:

- We turn the array into a heap
- We extract the largest element
- We apply recursion by sorting the remaining elements

Instead of using the preceding divide-and-conquer approach, it is better to use a dynamic programming approach. When we extract the largest element, we swap it with the last element of the array and make the heap one element shorter. The new, shorter heap does not need a full `build_heap` step because the only element out of order is the root node. We can fix this by a single call to `heapify`.

This is a possible implementation for the `heapsort` [15]:

```
def heapsort(A):
    heapify(A)
    n = len(A)
    for i in xrange(n-1,0,-1):
        (A[0],A[i]) = (A[i],A[0])
        heapify_one(A,0,i)
```

In the average and worst cases, it runs as fast as the quicksort, but in the best case, it is linear:

$$T_{best} \in \Theta(n) \tag{3.62}$$
$$T_{average} \in \Theta(n \log n) \tag{3.63}$$
$$T_{worst} \in \Theta(n \log n) \tag{3.64}$$
$$T_{memory} \in \Theta(1) \tag{3.65}$$

A heap can be used to implement a priority queue, for example, storage from which we can efficiently extract the largest element.

All we need is a function that allows extracting the root element from a heap (as we did in the `heapsort` and `heapify` of the remaining data) and a function to push a new value into the heap:

```
def heap_pop(A):
    if len(A)<1:
        raise RuntimeError('Heap Underflow')
    largest = A[0]
    A[0] = A[len(A)-1]
    del A[len(A)-1]
    heapify_one(A,0)
    return largest

def heap_push(A,value):
    A.append(value)
    i = len(A)-1
    while i>0:
        j = heap_parent(i)
        if A[j]<A[i]:
            (A[i],A[j],i) = (A[j],A[i],j)
        else:
            break
```

The running times for `heap_pop` and `heap_push` are the same:

$$T_{best} \in \Theta(1) \tag{3.66}$$

$$T_{average} \in \Theta(\log n) \tag{3.67}$$

$$T_{worst} \in \Theta(\log n) \tag{3.68}$$

$$T_{memory} \in \Theta(1) \tag{3.69}$$

Here is an example:

```
>>> a = [6,2,7,9,3]
>>> heap = []
>>> for element in a: heap_push(heap,element)
>>> while heap: print(heap_pop(heap))
9
7
6
3
2
```

Heaps find application in many numerical algorithms. In fact, there is a built-in Python module for them called `heapq`, which provides similar functionality to the functions defined here, except that we defined a max heap (pops the max element) while `heapq` is a min heap (pops the minimum):

```
>>> from heapq import heappop, heappush
>>> a = [6,2,7,9,3]
>>> heap = []
>>> for element in a: heappush(heap,element)
>>> while heap: print(heappop(heap))
9
7
6
3
2
```

Notice `heappop` instead of `heap_pop` and `heappush` instead of `heap_push`.

3.6.2 Binary search trees

A binary tree is a tree in which each node has at most two children (left and right). A binary tree is called a *binary search tree* if the value of a node is always greater than or equal to the value of its left child and less than or equal to the value of its right child.

A binary search tree is a kind of storage that can efficiently be used for searching if a particular value is in the storage. In fact, if the value for which we are looking is less than the value of the root node, we only have to search the left branch of the tree, and if the value is greater, we only have to search the right branch. Using divide-and-conquer, searching each branch of the tree is even simpler than searching the entire tree because it is also a tree, but smaller.

This means that we can search simply by traversing the tree from top to bottom along some path down the tree. We choose the path by moving down and turning left or right at each node, until we find the element for which we are looking or we find the end of the tree. We can search $T(d)$, where d is the depth of the tree. We will see later that it is possible to build binary trees where $d = \log n$.

To implement it, we need to have a class to represent a binary tree:

```
class BinarySearchTree(object):
    def __init__(self):
```