15-714: Resource Aware Programming Languages

Lectures 1 & 2: Traditional Complexity Analysis

Jan Hoffmann

August 29, 2025

1 Introduction

In the abstract, the goal of *Computer Science* is to understand the nature of computation and information. In practice, the goal is to create software and hardware systems that are both *correct* and *efficient*. There is a tension between correctness and efficiency that lies at the heart of our discipline: It is usually easier to write correct and inefficient code than correct and efficient code. The reason is that inefficient code tends to be simpler, while complex efficient code provides more opportunities for mistakes. Donald Knuth wrote that *premature optimization is the root of all evil*. Similarly, one could say that the pursuit of efficiency is the root of most bugs.

Computer Science can be organized into different subareas such as applications (machine learning, computer graphics, robotics, etc.), systems (operating systems, computer networks, computer architecture, etc.) theory (algorithm, computational complexity, cryptography, etc.), and programming languages (type theory, verification, compiler design, etc.). Naturally, correctness and efficiency are central in each of the subareas. However, among the three core areas, systems and theory have historically focused more on efficiency, whereas programming languages has focused more on correctness.

Within programming languages, efficiency has been mainly a systems concern. In other words, the focus has been on creating practical tools such as compilers that produce efficient code but not on providing information about the efficiency of the code to a programmer. For instance, a compiler rejects a program because it is likely incorrect, for example, as the result of a type error, but not because it is potentially inefficient. Type systems and program logic focus on functional properties of code and static analyses on finding bugs that compromise the security or correctness of a system rather than its performance.

This book is about *resource aware* programming languages.

> Resource aware programming languages provide *information* about the *resource consumption* of programs at development time.

This information can come in the form of concrete, non-asymptotic bounds on the time, memory, or energy consumption of a function or, more abstractly, as the guarantee that the implemented function is a member of a certain complexity class such as the functions computable in polynomial-time.

Analysis of Algorithms vs. Analysis of Programs To contrast resource-aware programming with the analysis of algorithms, it is helpful to first identify some of the applications of resource aware programming languages. The goals of these languages include to

1. ensure safety in resource-constraint systems such as embedded, real-time, or intermittent systems,
2. provide information to the user of a software library,
3. help write efficient code and avoid performance bugs,
4. simplify scheduling in cloud computing,

5. ensure security by mitigating side channels and algorithmic complexity attacks,

6. aid verification by enforcing cost bounds in smart contracts and complexity requirements in cryptographic protocols.

In analysis of algorithms, the goal is to understand an algorithm and its performance characteristics. This is important to select the best algorithm for a given task or to design new algorithms. However, the analysis of algorithms focuses on asymptotic bounds, which is unsuitable for many of the aforementioned applications. There are two reasons for the focus on asymptotic bounds. The first is the use of pseudo code, which is supposed to showcase the algorithm without distraction by implementation details. The lack of a formal syntax and semantics of the code makes it impossible to precisely define the resource cost of running the algorithm on a given input. Consequently, it does not make sense to derive bounds with constant factors.

The second reason is the heavy use of recurrence relations. The analysis of an algorithm is often performed in three steps: (1) identify a size measure for the inputs, (2) derive a recurrence relation that recursively defines the cost as a function of the size measure, and (3) find a closed form that is a bound for the recurrence. I argue that deriving a recurrence can be as hard as solving it, but commonly it is argued that finding a closed form is usually the most difficult part of the analysis. Most of the popular recipes for finding closed forms, like the *master method*, only yield functions that asymptotically bound the recurrence.

In the analysis of *programs* that resource aware languages provide, we are instead interested in concrete constant factors. Additionally, we also study programming languages that reject programs that are not *efficient*, that is, programs that potentially do not correspond to functions that are part of a certain complexity class like the functions computable in polynomial time.

2 Recurrence Relations

A popular (and somewhat systematic) way of performing the analysis of an algorithm is to use recurrence relations. Most commonly, recurrence relations are used for manual analyses but we also look at them from the perspective of automatic and mechanized resource analysis of programs.

First, we should define what a recurrence relation is. The textbook *Introduction to Algorithms* [CSRL01] contains the following definition.

> A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

One could argue that this definition is a bit too broad because an input does not necessarily have a unique order to which *smaller* can refer. Moreover, the aforementioned book contains only recurrence relations where the inputs are natural numbers. The definition from Carnegie Mellons's algorithms course (15-451/651) reads as follows.

> A recurrence relation is a description of the running time on an input of size n as a function of n and the running time on inputs of smaller sizes.

Here, we have the mention of a size but it is not clear if multiple arguments are permitted and if they all need to be abstracted to one size. The exact definition of the term recurrence relation is probably not that important. For the purpose of this course, we use the following one.

Definition. *A recurrence relation is a recursive function T of type $\mathbb{N}^k \to \mathbb{N}$.*

The idea is that the arguments of $T : \mathbb{N}^k \to \mathbb{N}$ represent different size measures that are related in some way to the inputs of the program that is the subject of the resource analysis. Resource analysis with recurrence relations proceeds in three steps. First, we identify the size measures that the bound should be a function of. Second, we derive a recurrence relation that describes the resource usage of the program in terms of the input sizes. Third, we find a closed (i.e., non-recursive) form of the recurrence. In the context of automatic or mechanized analysis, the two final steps are often called extracting and solving of recurrences.

Both deriving and solving recurrence relations are non-trivial. To start with, it is sometimes also not straightforward to find the right size measures for the analysis and an automatic approach would likely settle for the most common ones like the size or height of a data structure. The extraction of the recurrence is particularly difficult if we have higher-order functions, complex data structures, or also require a proof that the derived recurrence correctly describes the resource use of the program. One of the difficulties is that it is often not obvious how to describe the size of the arguments of recursive calls as a function of the original arguments. For example, these arguments could be returned by a complex auxiliary function before being passed to a recursive call. Finally, solving (i.e., finding closed forms for) recurrences is undecidable in general and also difficult to do manually. If we are looking for exact (non-asymptotic) solutions, we can only rely on a few techniques from calculus (see below) and mainly have to guess and substitute solutions or solution templates. The situation looks slightly better for asymptotic solutions and this is one of the reasons why asymptotic resource analysis is popular. However, even in the asymptotic case, solution recipes cover only specific cases such as divide-and-conquer (Master method).

2.1 The Substitution Method

We now perform a resource analysis for a simple program using recurrence relations in two steps:

1. We derive a recurrence relation $T(\vec{n})$ from the description of the algorithm (or the program).
2. We find a closed form for $T(\vec{n})$, that is, a non-recursive, easily-understood function $f(n)$ such that $f(n) = T(n)$.

To find such a function $f(n)$, we use the *substitution method*, which is one of the simplest and most general ways of solving recurrences.

Consider the factorial function *fac* that is implemented in OCaml below.

```
let rec fac n =
  if n > 1 then
    n * (fac (n-1))
  else
    1
```

Assume we are interested in the number of multiplications that are performed by *fac*. The first question that arises is how to abstract an integer with a natural number. In this case, we can see right away that the cost for negative inputs is 0 and focus on positive integers, which are simply abstracted by their values. We can express the cost with the following recurrence.

$$\begin{aligned} T_{\text{fac}}(0) &= 0 \\ T_{\text{fac}}(1) &= 0 \\ T_{\text{fac}}(n) &= T_{\text{fac}}(n-1)+1 \quad \text{if } n > 1 \end{aligned}$$

If $n = 0$ or $n = 1$ then the program does not execute any multiplications. If $n > 1$ then *fac(n)* performs a multiplication plus the multiplications performed by *fac(n-1)*.

Next, we solve the recurrence relation with the *substitution method*. We guess the solution $f(n)$ for $T_{\text{fac}}(n)$ is linear, that is, $f(n) = c_1 n + c_0$ for constants c_1 and c_0. To verify that $f(n)$ is a correct solution and to determine the constants, we substitute $f(n)$ into the recurrence relations. We obtain

$$\begin{aligned} 0 \cdot c_1 + c_0 &= 0 \\ 1 \cdot c_1 + c_0 &= 0 \end{aligned}$$

which implies $c_0 = c_1 = 0$. However, this doesn't satisfy the remaining equality $T_{\text{fac}}(n) = T_{\text{fac}}(n-1)+1$. We conclude that there does not exist a linear solution for the recurrence relation.

We could now start looking for non-linear solutions but the recurrence is so simple that we can see that the problem originates from the base cases. So we simply drop $T_{\text{fac}}(0) = 0$ and only

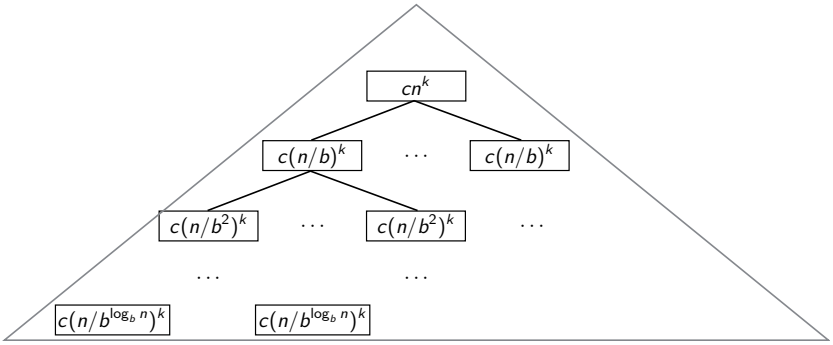

Figure 1: Recursion tree for $T(n) = aT(\frac{n}{b}) + cn^k$.

focus on positive numbers. While this seems to be an insignificant detail in our manual analysis, such issues make it difficult to automatically derive the right recurrence relations from code.

We again substitute our guess (or *ansatz*) into the remaining two equations.

$$\begin{aligned} c_1 + c_0 &= 0 \\ c_1(n+1) + c_0 &= c_1 n + c_0 + 1 \end{aligned}$$

Now we can solve for c_1 and c_0 and obtain $c_1 = 1$ and $c_0 = -1$. A sanity check shows that $f(n) = n - 1$ is indeed a solution of the recurrence.

Unfortunately, coming up with a candidate for a solution is not always straightforward.

2.2 The Recursion-Tree Method

A powerful method for finding an asymptotic solution (or candidate solution) for a recurrence for particular divide-and-conquer algorithms is the *recursion tree method* (or master method). Recall the following definitions for asymptotic notation.

Definition 1. *Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be two functions from natural numbers to non-negative reals. We write*

- $f \in O(g)$ *if there exists $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$*
- $f \in \Omega(g)$ *if there exists $c > 0$ and $n_0 \in \mathbb{N}$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$*
- $f \in \Theta(g)$ *if $f \in \Omega(g)$ and $f \in O(g)$*

You may notice that these definitions only apply to functions with one argument and it is not trivial to generalize them to multiple arguments.

Consider a divide-And-conquer algorithm that divides a problem of size n into a sub-problems of size n/b, which are recursively solved. The solutions of the sub-problems are then combined to a solution of the original problem of size n. Assume that the combination of the sub-solutions and the division into sub-problems needs time cn^k. The run time of such an algorithm can then be expressed as a recurrence relation $T(n)$ as follows where a,b,c, and k are positive constants.

$$\begin{aligned} T(1) &= c \\ T(n) &= aT(\tfrac{n}{b}) + cn^k \quad \text{if } n > 1 \end{aligned}$$

Example 1. *Consider for example the divide-and-conquer algorithm* merge sort. *Merge sort divides a problem of size n into 2 sub-problems of size $\frac{n}{2}$. Division and merging takes time cn for some constant c. We thus have $a = b = 2$, $k = 1$ and $T_{ms} = 2T_{ms}(\frac{n}{2}) + cn$.*

The recurrence relation $T(n) = aT(\frac{n}{b}) + cn^k$ can be solved systematically using the recursion-tree method. To see how, consider the recursion tree that arises when unrolling $T(n)$ in Figure 1. We derive a closed form for $T(n)$ by summing up the cost level by level. The cost for the top level

is cn^k, the cost for the second level is $ac(\frac{n}{b})^k$, and the cost of the third level is $a^2c(\frac{n}{b^2})^k$. In total, the sum of costs is

$$T(n) = cn^k\left(1 + \frac{a}{b^k} + \left(\frac{a}{b^k}\right)^2 + \left(\frac{a}{b^k}\right)^3 + \cdots + \left(\frac{a}{b^k}\right)^{\log_b n}\right) \tag{1}$$

If we define $r = \frac{a}{b^k}$ then we have

$$T(n) = cn^k\left(1 + r + r^2 + r^3 + \cdots + r^{\log_b n}\right) \tag{2}$$

We now distinguish three cases.

Case $r < 1$**.** Then $1 + r + r^2 + r^3 + \cdots + r^{\log_b n}$ are the first $\log_b n$ terms of a geometric series. So

$$1 + r + r^2 + r^3 + \cdots + r^{\log_b n} < \sum_{i=0}^{\infty} r^i = \frac{1}{1-r}$$

and $cn^k \le T(n) \le \frac{cn^k}{1-r}$. Thus $T(n) \in \Theta(n^k)$ since r and c are constants.

Case $r = 1$ Then $1 + r + r^2 + r^3 + \cdots + r^{\log_b n} = \log_b n + 1$ and

$$T(n) = cn^k(\log_b n + 1) \in \Theta(n^k \log n)\,.$$

Case $r > 1$ Then we have still the first $\log_b n$ terms of a geometric series and thus

$$1 + r + r^2 + r^3 + \cdots + r^{\log_b n} = \frac{1 - r^{\log_b n + 1}}{1 - r} = \frac{r^{\log_b n + 1} - 1}{r - 1} = \Theta(r^{\log_b n})\,.$$

Thus

$$\begin{array}{rcll} T(n) & \in & \Theta(n^k(\frac{a}{b^k})^{\log_b n}) & \\ & = & \Theta(a^{\log_b n}) & \text{since } b^{k\cdot\log_b n} = n^k \\ & = & \Theta(n^{\log_b a}) & \text{since } a^{\log_b n} = b^{(\log_b a)(\log_b n)} = n^{\log_b a} \end{array}$$

In summary, the so called *master method* for solving recurrences for divide-and-conquer algorithms is stated by the following theorem.

Theorem 1 (Master Theorem)**.** *Let* $T(n) = aT(\frac{n}{b}) + cn^k$ *be a recurrence relations and let* $a, b, c, k > 0$ *be positive constants. Then*

$$T(n) \in \begin{cases} \Theta(n^k) & \text{if } \frac{a}{b^k} < 1 \\ \Theta(n^k \log n) & \text{if } \frac{a}{b^k} = 1 \\ \Theta(n^{\log_b a}) & \text{if } \frac{a}{b^k} > 1 \end{cases}$$

There are many variants and generalizations of the recursion tree method and more powerful "master theorems" in which cn^k is replaced with an arbitrary function $f(n)$. However, they are all limited to divide-and-conquer algorithms, derive asymptotic bounds only, have only one parameter, and cannot be applied if the "split factors" a and b are not constants.

Example 2. *Consider again* merge sort *with recurrence* $T_{ms}(n) = 2T_{ms}(\frac{n}{2}) + cn$ *and* $a = b = 2$*, and* $k = 1$*. Therefore* $\frac{a}{b^k} = 1$ *and by Theorem 1 it follows that* $T_{ms}(n) = \Theta(n \log n)$*.*

So how useful is the recursion tree method for finding an *ansatz* for exactly solving a recurrence relation? It is only partially helpful. Do not be deceived by the use of Θ in the theorem. If $T(n) \in \Theta(n^k)$ does not mean that $T(n) = cn^k$ for some c. It's probably worth trying an ansatz with a polynomial of degree k. However, arbitrary functions like $n^k + n \log n$ are members of $\Theta(n^k)$. So there is no guarantee that this approach will succeed.

```
let rec split l =
  match l with
    | [] → ([],[])
    | x1::xs →
      match xs with
        | [] → ([x1],[])
        | x2::xs' →
          let (l1,l2) = split xs' in
          (x1::l1,x2::l2)
```

Figure 2: The split function of merge sort.

2.3 Linear Recurrence Relations

One of the few techniques for obtaining exact solutions for recurrence relations is based on calculating roots of characteristic polynomials. It applies to *homogeneous linear recurrence relations with constant coefficients*. These are recurrence relations of the form

$$T(n) = c_1 T(n-1) + \cdots + c_d T(n-d)$$

for constants c_i with $c_d \neq 0$. We call d the degree of the recurrence. To uniquely define the value of the recurrence T we have to define the initial or base values $T(0), \ldots, T(d-1)$.

We can find a closed form for a homogeneous linear recurrence relation by finding the roots of the characteristic polynomial

$$P(x) = x^d - c_1 x^{d-1} - \cdots - c_d$$

which has d roots $r_1, \ldots, r_d$. There are two cases. If the roots are pairwise distinct then

$$T(n) = k_1 r_1^n + \ldots + k_d r_d^n$$

for constants k_i that can be determined using the initial values $T(0), \ldots, T(d-1)$.

If the roots are not pairwise distinct then we add additional terms to the products in the solution that are determined by the number of identical roots that occurred in the formula already:

$$T(n) = k_1 n^{u_1} r_1^n + \ldots + k_d n^{u_d} r_d^n$$

Here, u_i is number of identical roots r_j ($j < i$ and $r_j = r_i$) that appeared before r_i in the formula. Note that this is a generalization of the formula for pairwise distinct roots.

Non-Homogeneous Recurrence Relations Recurrence relations that arise in resource analysis are usually not homogeneous because they have a cost component in addition to the cost of the base cases. The form of such *non-homogeneous linear recurrence relations with constant coefficients* is given by

$$T(n) = c_1 T(n-1) + \cdots + c_d T(n-d) + P(n)$$

where $P(n)$ is a function in n. If $P(n)$ is a polynomial then we can reduce the problem of finding a solution to the problem of finding a solution for a homogeneous linear recurrence relation.

This is best illustrated with an example. Let us consider the split function for merge sort in Figure 2, which recursively splits a list into two lists of (almost) equal length. To this end, split removes the first two list elements, recursively splits the remaining list, and adds one of the removed elements to each of the returned lists. If we count the number of cons operations, the cost can be described by the following recurrence relation.

$$\begin{array}{lcl} T_{\text{split}}(0) & = & 0 \\ T_{\text{split}}(1) & = & 1 \\ T_{\text{split}}(n+2) & = & 2 + T_{\text{split}}(n) \end{array}$$

```
let rec append l1 l2 =
  match l1 with
  | [] → l2
  | x::xs → x::(append xs l2)

let rec partition f l =
  match l with
  | [] → ([],[])
  | x::xs →
     let (cs,bs) = partition f xs in
     if f x then
       (cs,x::bs)
     else
       (x::cs,bs)

let rec quicksort le = function
  | [] → []
  | x::xs →
     let ys, zs = partition (le x) xs in
     append (quicksort le ys) (x :: (quicksort le zs))
```

Figure 3: An implementation of quicksort in OCaml.

To convert the recurrence into a homogeneous recurrence that has the same closed-form solution, we use the equality $T_{\text{split}}(n+3) = 2 + T_{\text{split}}(n+1)$ to subtract $T_{\text{split}}(n+3)$ from both sides of the original recurrence. We obtain $T_{\text{split}}(n+2) - T_{\text{split}}(n+3) = 2 + T_{\text{split}}(n) - 2 - T_{\text{split}}(n+1)$ and thus

$$T_{\text{split}}(n) \quad = \quad T_{\text{split}}(n-1) + T_{\text{split}}(n-2) - T_{\text{split}}(n-3)$$

The characteristic polynomial for $T_{\text{split}}(n)$ is

$$P_{\text{split}}(x) = x^3 - x^2 - x + 1$$

The roots of the characteristic polynomial are 1 and -1. The root 1 appears twice as we can write $P_{\text{split}}(x)$ as $(x-r)^2 q(x)$ for $r = 1$ and a polynomial $q(x)$, namely $q(x) = x + 1$.

Using the previously discussed formula we have

$$\begin{aligned} T_{\text{split}}(n) &= k_1(-1)^n + k_2 1^n + k_3 n 1^n \\ &= k_1(-1)^n + k_2 + k_3 n \end{aligned}$$

for yet undetermined constants k_i. Since $T_{\text{split}}(0) = 0$ we have $k_1 + k_2 = 0$. Since $T_{\text{split}}(1) = 1$ we have $k_2 + k_3 - k_1 = 1$ and since $T_{\text{split}}(2) = 1$ we have $k_1 + k_2 + 2k_3 = 2$. Therefore $k_1 = k_2 = 0$, $k_3 = 1$, and $T_{\text{split}}(n) = n$.

We can also experiment with other cost. If we for example define $T_{\text{split}}(1) = 0$ then we get the more interesting result $T_{\text{split}}(n) = 0.5(-1)^n - 0.5 + n$.

2.4 Example: Quick Sort

Hoare's quicksort algorithm is an example of a divide-and-conquer algorithm for which the master method cannot be applied. Figure 3 shows an implementation of quicksort in OCaml.

Our goal is to use recurrence relations to analyze the worst-case complexity of quicksort. The cost model we consider is the number of function calls. The first challenge in the analysis is that quicksort is higher-order function: It takes the comparison function *le* as its first argument. This is not uncommon, even in languages like C which do not have first class functions such as OCaml. In the C standard library, one of quicksort's arguments is a pointer to a comparison function. We ignore the issue of higher-order functions for now and just assume that all comparison functions have a constant cost. The second challenge in the analysis is that quicksort is a curried function:

The call *quicksort gt* consumes a constant amount of resources and returns a function closure. We will also ignore the issue of curried functions and will for now assume that quicksort is always applied to both of its arguments.

Before we can analyze quicksort, we have to analyze the helper functions *append* and *partition*. The recurrence relations that we obtain from the code are

$$\begin{array}{lcll} T_{\text{app}}(0,m) & = & 0 & \\ T_{\text{app}}(n,m) & = & T_{\text{app}}(n-1,m)+1 & \text{if } n>1 \end{array}$$

$$\begin{array}{lcll} T_{\text{par}}(0) & = & 0 & \\ T_{\text{par}}(n) & = & T_{\text{par}}(n-1)+2 & \text{if } n>1 \end{array}$$

For *append* (T_{app}), n is the length of the first list in the argument and m is the length of the second list in the argument. For *partition* (T_{par}), n is the length of the second argument. In the second equation of par we count the recursive call and the call to the function f. Similar as for the recurrence T_{fac}, we can derive $T_{\text{app}}(n,m) = n$ and $T_{\text{par}}(n) = 2n$.

We proceed to derive a recurrence relation T_{qs} for the function *quicksort*. When looking at the code, the question arises what the size of the lists *ys* and *zs* in the recursive calls for quicksort is. In fact, if $|\ell|$ denotes the length of a list ℓ, we have $|ys| + |zs| = |xs|$. So we need to prove by induction on *xs* that for every function *f*: if *(ys,zs) = partition f xs* then $|ys| + |zs| = |xs|$.

Before we can state the recurrence for *quicksort*, we also have to show that $|$*quicksort le xs*$| = |xs|$. However, we omit this step for brevity.

$$\begin{array}{rcl} T_{\text{qs}}(0) & = & 0 \\ T_{\text{qs}}(n+1) & = & \max_{0\le i\le n}\left(T_{\text{qs}}(i)+T_{\text{qs}}(n-i-1)+i\right)+2n+4 \end{array}$$

Let us first assume that the array is always split in the middle, that is, $i = \lfloor n/2 \rfloor$. We then have

$$T_{\text{qs}}(n) = 2\,T_{\text{qs}}(n/2) + \Theta(n)$$

and can use case $a/b^k = 1$ of the master method to infer $T_{\text{qs}}(n) \in \Theta(n \log n)$.

Let us now assume that the list is always split in the most imbalanced way, that is, $i = 0$. Then

$$\begin{array}{rcl} T_{\text{qs}}(n) & = & T_{\text{qs}}(n-1) + T_{\text{qs}}(0) + \Theta(n) \\ & = & T_{\text{qs}}(n-1) + \Theta(n) \end{array}$$

In this case, We can show with the substitution method that $T_{\text{qs}}(n) \in \Theta(n^2)$.

We conjecture that in general $T_{\text{qs}}(n) \le cn^2$ for a constant c and use the substitution method to prove it.

$$\begin{array}{rcl} T_{\text{qs}}(n) & \le & \max_{0\le i\le n-1}(ci^2 + c(n-i-1)^2)) + \Theta(n) \\ & = & \max_{0\le i\le n-1} c(i^2 + (n-i-1)^2)) + \Theta(n) \end{array}$$

For a fixed n, the polynomial $i^2 + (n-i-1)^2)$ takes maximums in the range $0 \le i \le n-1$ for $i = 0$ and $i = n-1$. To verify this claim observe that the second derivative with respect to i is positive. Thus we have

$$\max_{0\le i\le n-1} i^2 + (n-i-1)^2) \le (n-1)^2 = n^2 - 2n + 1$$

and conclude

$$T_{\text{qs}} \le cn^2 - 2cn + c + \Theta(n) \in O(n^2)\,.$$

We have now performed only an asymptotic analysis. To determine the exact constants is even more tedious. It is not even clear how an exact solution to the recurrence could look like since, as we have seen, the resource usage varies based on the choices of the pivot element. However, we have abstracted away the elements of the input list and it seems impossible to recover from that abstraction. So we stop here as the previous calculations already illustrate the difficulties with automating and mechanizing recurrence solving.

3 Amortized Analysis with the Potential Method

Amortized analysis is a technique to derive a worst-case resource bound for a sequence of operations (or function calls). For many data structures, the amount of resources that an operation consumes can vary substantially depending on the state of the data structure. Nevertheless, high cost (e.g., reorganizing the data structure) will occur with some predictable frequency; they *amortize* over time. Summing up the worst-case costs of operations in such a sequence of operations would lead to gross over-approximations of the cost. In these cases, amortized analysis provides a much tighter resource bound.

Remark. *You sometimes read that amortized analysis is a way of determining the amortized or even* average *cost of an operation. However, the goal of an amortized analysis is to determine the worst-case cost of a sequence of operations. As a means to an end, we define the amortized cost of an operation.*

Potential Functions To amortize the cost of different operations, we introduce a potential function

$$\Phi : \text{State} \to \mathbb{Q}_{\geq 0} .$$

The idea is that an operation $o : \text{State} \to \text{State}$ that is executed in state S can use the potential $\Phi(S)$ to pay for the cost of the operation. More specifically, we say that the *amortized cost* of operation o in state S with respect to Φ is

$$acost(o) = \text{cost}(o(S)) + \Phi(o(S)) - \Phi(S)$$

where $\text{cost}(o(S))$ is the actual resource consumption of o in state S.

Theorem 2. *Given starting state S_1 and operations $o(S_1), \ldots, o(S_n)$ such that $o(S_i) = S_{i+1}$, we have*

$$\sum_{1 \leq i \leq n} cost(o(S_i)) \leq \sum_{1 \leq i \leq n} acost(o(S_i)) + \Phi(S_1) .$$

Proof. We write o_i for $o(S_i)$.

$$\begin{array}{lcl} \sum_{1 \leq i \leq n} acost(o_i) + \Phi(S_1) & \geq & \Phi(S_1) + \sum_{1 \leq i \leq n} \text{cost}(o_i) + \Phi(S_{i+1}) - \Phi(S_i) \\ & = & \Phi(S_1) + \Phi(S_{n+1}) - \Phi(S_1) + \sum_{1 \leq i \leq n} \text{cost}(o_i) \\ & = & \Phi(S_{n+1}) + \sum_{1 \leq i \leq n} \text{cost}(o_i) \\ & \geq & \sum_{1 \leq i \leq n} \text{cost}(o_i) \end{array}$$

□

The challenge in performing an amortized analysis is to choose the right potential function Φ. The goal is to choose Φ so that the amortized cost $\text{cost}(o(S)) + \Phi(o(S)) - \Phi(S)$ is similar for every state S.

Example: Stack Given is an implementation of a stack with operations *push* and *pop*. We assume that $\text{cost}(push(S,x)) = 1$ and $\text{cost}(pop(S)) = \min(|S|, 1)$. Consequently, the cost of a sequence of n *push* and *pop* operations is less or equal to n.

Consider the additional operation *multipop(S,k)* that pops k elements from the stack. If *multipop(S,k)* is executed for a stack S with less than k elements then all elements of S are popped. Similarly, if *pop(S)* is applied to an empty stack S then S is unchanged and some error message is returned. The cost of *multipop(S,k)* is $\min(|S|, k)$.

What is the worst-case cost of a sequence of n *push*, *pop*, and *multipop* operations? In a conservative analysis we would first establish the worst-case cost of each operation. So we would argue that the worst-case cost for *multipop* is $n-1$ since there are $n-1$ elements on the stack in the worst-case. Consequently, the cost of n operations can only be bounded by n^2. This is a very loose bound since every element on the stack can only be popped once.

Using the potential method, we can perform a much more precise analysis. For a stack S we define the potential

$$\Phi(S) = |S|$$

to be the height of the stack. The amortized cost of push is then

$$\mathit{acost}(\mathit{push}(S,x)) = 2$$

since $\Phi(\mathit{push}(S,x)) - \Phi(S) = 1$ for every stack S and $\mathrm{cost}(\mathit{push}(S)) = 1$. Furthermore, we have $\Phi(\mathit{pop}(S)) - \Phi(S) = -\min(|S|,1)$ and $\mathrm{cost}(\mathit{pop}(S)) = \min(|S|,1)$ for every S and thus

$$\mathit{acost}(\mathit{pop}(S)) = 0\,.$$

Similarly, $\Phi(\mathit{multipop}(S,k)) - \Phi(S) = -\min(|S|,k)$ and $\mathrm{cost}(\mathit{multipop}(S,k)) = \min(|S|,k)$ for every S and thus

$$\mathit{acost}(\mathit{multipop}(S,k)) = 0\,.$$

Therefore it follows from Theorem 2 that the cost of n operations is at most $2n$. It is also easy to see that the cost is actually bounded by $2m$, where m is the number of *push* operations in the sequence.

Example: Binary Counter Consider a binary counter $b = b_k, \ldots, b_0$, which is implemented using a list or an array of bits of fixed length. The binary counter only has an operation $\mathit{inc}(b)$, which increments b by 1. The cost of $\mathit{inc}(b)$ is defined as the number of bits that have to be flipped for the update. We observe that at most one 0 is flipped to a 1 in an increment. However, multiple 1s can be flipped to 0s. Let $|b|_1$ be the number of 1s in the counter. Then $|b|_1 + 1$ is an upper bound on the number of bits that are flipped by one increment.

We are interested in the cost of n increment operations. To derive a worst-case bound, we use the potential method of amortized analysis. We define the potential

$$\Phi(b) = |b|_1$$

To determine the *amortized cost* $\mathit{acost}(\mathit{inc}(b))$, assume that the operation $\mathit{inc}(b)$ modifies t_i bits. As discussed earlier, $t_i - 1$ flips are from 1 to 0 and at most one flip (zero can happen during overflow) is from 0 to 1. So we have

$$\begin{aligned} \mathrm{cost}(\mathit{inc}(b)) + \Phi(\mathit{inc}(b)) - \Phi(b) &= t_i + \Phi(\mathit{inc}(b)) - \Phi(b) \\ &\leq t_i + (\Phi(b) - (t_i - 1) + 1) - \Phi(b) \\ &= 2 \end{aligned}$$

It follows that $\Phi(b_0) + 2n$ is an upper bound on the number of flips in a sequence of n increment operations. Here b_0 is the initial counter.

Example: Dynamic Table Another standard example of amortized analysis is a dynamic table T that is implemented with an array. Assume first that we only have an insert operation $\mathit{insert}(T,x)$, which inserts a new element x into the table. We simply insert the new elements successively until the array is full. In this case, we allocate a new array of double the size of the current array and copy the elements over. Our cost model in this example is to count the number of writes to an array. So the worst-case cost of one operation $\mathit{insert}(T,x)$ is $|T| + 1$. However, we are again interested in a sequence of n insert operations.

Let $|T|_{\mathrm{elm}}$ be the number of stored elements in T. We define the potential function

$$\Phi(T) = 2|T|_{\mathrm{elm}} - |T|$$

If we always start with an empty array then the load factor is never smaller than $1/2$ and $\Phi(T) \geq 0$. If we consider arbitrary load factors at the beginning, we can instead define $\Phi(T) = \max(0, 2|T|_{\mathrm{elm}} - |T|)$.

We now determine the amortized cost. There are two cases. Assume first that $|T|_{\text{elm}} = |T|$. Then

$$\begin{aligned} \text{cost}(\mathit{insert}(T,x)) + \Phi(\mathit{insert}(T,x)) - \Phi(T) &= |T| + 1 + \Phi(\mathit{insert}(T,x)) - (2|T|_{\text{elm}} - |T|) \\ &= |T| + 1 + \Phi(\mathit{insert}(T,x)) - |T| \\ &= |T| + 1 + (2(|T|+1) - 2|T|) - |T| \\ &= 3 \end{aligned}$$

Assume now that $|T|_{\text{elm}} < |T|$. Then

$$\begin{aligned} \text{cost}(\mathit{insert}(T,x)) + \Phi(\mathit{insert}(T,x)) - \Phi(T) &= 1 + \Phi(\mathit{insert}(T,x)) - (2|T|_{\text{elm}} - |T|) \\ &= 1 + (2|T|_{\text{elm}} + 2 - |T|) - (2|T|_{\text{elm}} - |T|) \\ &= 3 \end{aligned}$$

As a result, we have $\mathit{acost}(\mathit{insert}) = 3$.

We now add another operation $\mathit{remove}(T,x)$ that deletes the element x from the table T. Similarly, the expansion now also want to contract the table if the load factor is equal to a constant $0 \le c \le 1$. You might have the intuition to pick $c = 1/2$. However, this is not a good choice since a sequence of alternating inserts and deletes can cause quadratic cost in the length of the sequence.

A better choice is $c = 1/4$. Now we define the potential to be

$$\Phi(T) = \begin{cases} 2|T|_{\text{elm}} - |T| & \text{if } |T|_{\text{elm}}/|T| \ge 1/2 \\ |T|/2 - |T|_{\text{elm}} & \text{if } |T|_{\text{elm}}/|T| < 1/2 \end{cases}$$

The amortized costs are then $\mathit{acost}(\mathit{insert}) = 3$ and $\mathit{acost}(\mathit{remove}) = 3$.

References

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley, Redwood City, CA, USA, 1997.