Concept: Run time analysis

The exposition is based on the following sources, which are all recommended reading:

1. Corman, Leiserson, Rivest: Introduction to algorithms, Chapter 18
2. Heun: Grundlegende Algorithmen, Chapter 1 and 2.4

Concept: Run time analysis (2)

In this Section we recall the basic notations for run time analyses and then describe the different concepts of *worst-case run time*, *average case run time*, *expected run time*, and *amortized run time*.

Lets start by recalling the definitions of the *Landau* symbols ($O, \Omega, \Omega_\infty, \Theta, o, \omega$).

Concept: Run time analysis (3)

$$O(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : \exists c \in \mathbb{R} > 0, n_0 \in \mathbb{N} : \forall n \in \mathbb{N}, n \geq n_0 : g(n) \leq c \cdot f(n)\} \quad (5.1)$$

$$\Omega(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : \exists c \in \mathbb{R} > 0, n_0 \in \mathbb{N} : \forall n \in \mathbb{N}, n \geq n_0 : g(n) \geq c \cdot f(n)\} \quad (5.2)$$

$$\Omega_\infty(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : \exists c \in \mathbb{R} > 0 : \forall m \in \mathbb{N} : \exists n \in \mathbb{N}, n > m : g(n) \geq c \cdot f(n)\} \quad (5.3)$$

$$\Theta(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : g \in O(f) \text{ and } g \in \Omega(f)\} \quad (5.4)$$

$$o(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0\} \quad (5.5)$$

$$\omega(f) := \{g : \mathbb{N} \to \mathbb{R}_+ : \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0\} \quad (5.6)$$

In the following we list some commonly used adjectives describing classes of functions. Mind that we use the more common = sign instead of the (more correct) $\in$ sign.

Concept: Run time analysis (4)

We say af function f:

- is *constant*, if $f(n) = \Theta(1)$
- grows *logarithmically*, if $f(n) = O(\log n)$
- grows *polylogarithmically*, if $f(n) = O(\log^k(n))$ for a $k \in \mathbb{N}$.
- grows *linearly*, if $f(n) = O(n)$
- grows *quadratic*, if $f(n) = O(n^2)$

Concept: Run time analysis (5)

We say af function f:

- grows *polynomially*, if $f(n) = O(n^k)$, for a $k \in \mathbb{N}$.
- grows *superpolynomially*, if $f(n) = \omega(n^k), \forall k \in \mathbb{N}$.
- grows *subexponentially*, if $f(n) = o(2^{cn}), \forall 0 < c \in \mathbb{R}$.
- grows *exponentially*, if $f(n) = O(2^{cn})$ for a $0 < c \in \mathbb{R}$.

Concept: Run time analysis (6)

After that reminder lets introduce the different run time definitions and explain them using an example.

- worst case analysis: We assume for both, *the input* and *the execution of the algorithm* the worst case. The latter is of course only applicable for non-deterministic algorithms.
- best case analysis: We assume for both, *the input* and *the execution of the algorithm* the best case. The latter is of course only applicable for non-deterministic algorithms.
- average case analysis: We average over all possible *input* the run time of our (deterministic) algorithm.

Concept: Run time analysis (7)

- expected run time analysis: Our algorithm runs depending on the value of some random variables for which we know their distributions. Hence we try to estimate the *expected* run time of the algorithm.
- amortized analysis: Sometimes, an algorithm (usually an operation on a data structure) needs a long time to run, but changes the data structure such that subsequent operations are not costly. A worst case run time analysis would be inappropriate. An *amortized* analysis averages over a series of operations (not over the input).

Example: quicksort (8)

The well-known (deterministic) quicksort algorithm for sorting an array chooses a fixed element as its pivot element, lets say w.l.o.g. the first one. It arranges all smaller elements on the left of the pivot, all larger ones on the right and recurses on the two halfs.

- worst case analysis: In the worst case, the left (or the right) half are always empty. Hence the worst case run time is the solution to the recurrence
  $f(n) = (n-1) + f(n-1), f(0) = 0$. Obviously $f = O(n^2)$.
- best case analysis: In the best case, the left and the right half differ in size by at most one. Hence the best case run time is the solution to the recurrence
  $f(n) = (n-1) + 2 * f(n/2), f(0) = 0$. Obviously $f = O(n \log n)$.
- average case analysis: We average over all possible *inputs* the run time of the deterministic quicksort. The result is that on average quicksort needs $O(n \log n)$ comparisons (exercise).

Example: quicksort (9)

The dependence n the input and the bad worst case run time of quicksort are worrysome. Quicksort (like many other algorithms) can be made considerably more robust by *randomizing* the algorithm. In randomized quicksort we choose the pivot element randomly using the value of a random variable uniformly distributed over $[1, n]$.

- expected run time analysis: randomized quicksort can be shown to run in *expected* time $O(n \log n)$ with *high probability*. We will discuss such an analysis in detail using skiplists later in the lecture.

We did not deal with *amortized analysis* in this example. We will introduce this concept now.

Concept: Run time analysis

In this Section we describe the concept of amortized analysis of operations on data structures. Quite often, operations have a bad worst case running time, however such a running time does not occur very *often*. Hence it would be unfair to judge the performance of this data structure using their worst case running time.

Instead we use the *average* run time over a series of n operations where the sequence of operations is as ”bad” as possible for the data structure, or put it differently we want a guarantee for the average performance in the worst case.

Note that this is different from the average case analysis, in which you average the run times of the operation over all possible inputs.

Concept: Amortized analysis

Imagine for example a stack. We have the following operations on the stack

- `Pop(S)` pops the top element of the stack and returns it.
- `Push(S,x)` pushed element x on the stack.
- `MultiPop(S,k)` returns at most the top k elements from the stack (it calls `Pop` k times).

Obviously the operations `Pop` and `Push` have worst case time $O(1)$. However the operation `MultiPop` can be linear in the stack size. So if we assume that at most n objects are on the stack a multipop operation can have worst case cost of $O(n)$. Hence in the worst case a series of n stack operations is bounded by $O(n^2)$.

We will now use this example to illustrate three different techniques to find a more realistic amortized bound for n operations.

Concept: Amortized analysis (2)

The three methods to conduct the analysis which are:

- The aggregate method
- The accounting method
- The potential method

The aggregate method

Here we show that for all n, a sequence of n operations takes worst-case time $T(n)$ in total. Hence, for this worst case, the average cost, or amortized cost is $T(n)/n$. Note that this method charges the same amortized cost to each operation in the sequence of operations, even if this sequence contains different types of operations.

The other two methods can assign individual amortized costs for each type of operation.

The aggregate method (2)

We argue as follows. In any sequence of n operations on an initially empty stack, each object can be popped at most once for each time it is pushed. Therefore, the number of times `Pop` can be called on a nonempty stack (including calls within `MultiPop`), is at most the number of `Push` operations, which is at most n.

Hence, for any value of n, any sequence of n `Push`, `Pop`, and `MultiPop` operations takes a total of $O(n)$ time. Hence the amortized cost of any operation is $O(n)/n = O(1)$.

The accounting method

In the accounting method we assign differing charges to different operations, with some operations charge more or less than they actually cost. The amount we charge an operation is called its *amortized* cost.

When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as *credit*. Credit can then be used to pay later for operations whose amortized cost is less than their actual cost.

The accounting method (2)

One must choose the amortized costs carefully. If we want the analysis with amortized costs to show that in the worst case the average cost per operation is small, then the total amortized cost of a sequence of operations must be an upper bound on the total actual cost of the sequence. Moreover, this relationship must hold for all sequences of operations, thus the total credit must be nonnegative at all times.

Let us return to our stack example.

The accounting method (3)

The actual costs are:

- `Pop(S)` 1,
- `Push(S,x)` 1,
- `MultiPop(S,k)` $\min(k, s)$, where s is the size of the stack.

Let us assign the following amortized costs.

- `Pop(S)` 0,
- `Push(S,x)` 2,
- `MultiPop(S,k)` 0.

The accounting method (4)

Note that the amortized costs of *all* operations is $O(1)$ and hence the amortized cost of n operations is $O(n)$. Note also that the actual cost of `MultiPop` is variable whereas the amortized cost is constant.

Using the same argument as in the aggregate method it is easy to see that our account is always charged starting with an empty stack. Each `Push` operation pays 2 credits. One for its own cost and one for the cost of popping the element off the stack, either through a normal `Pop` or through a `MultiPop` operation.

Please note that this method can assign individual, different amortized costs to each operation.

The potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the *potential method* represents the prepaid work as "potential energy" or simply "potential" that can be released to pay for future operations.

The potential is associated with the data structure as a whole rather than with specific objects within the data structure. It works as follows. We start with an initial data structure D_0, on which n operations are performed. For each $i = 1, 2, \ldots, n$ we let c_i be the actual cost of the i-th operation and D_i be the data structure that results after applying the i-th operation.

The potential method (2)

A *potential function* Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the potential associated with the data structure D_i. The amortized cost $\hat{c}_i$ of the i-th operation with respect to the potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

that is, the amortized cost is the actual cost plus the increase in potential due to its operation. Hence the total amortized cost of the n operations is:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

The potential method (3)

If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ is an upper bound on the total actual cost. In practice we do not know how many operations might be performed and therefore guarantee the $\Phi(D_i) \geq \Phi(D_0)$, for all i. It is often convenient to define $\Phi(D_0) = 0$ and then show that all other potentials are non negative.

Lets illustrate the method using our stack example.

The potential method (4)

We define the potential function on the stack as the number of elements it contains. Hence the empty stack D_0 has $\Phi(D_0) = 0$. Since the number of objects on the stack is never negative we have $\Phi(D_i) \geq 0$ for all stacks D_i resulting after the i-th operation.

Let us now compute the amortized costs of the various stack operations. If the i-th operation on a stack containing s objects is a `Push` operation, then the potential difference is

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1 \end{aligned} \tag{5.7}$$

Hence the amortized cost is:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2.$$

The potential method (5)

If the i-th operation on a stack containing s objects is a `MultiPop(S,k)` then $k' = \min(k, s)$ objects are popped off the stack. The actual cost of the operation is k' and the potential difference is:

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s - k') - s \\ &= -k' \end{aligned} \tag{5.8}$$

Hence the amortized cost is:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0.$$

A similar result is obtained for `Pop`. This shows that the amortized cost of each operation is $O(1)$ and hence the total amortized cost of a sequence of n operations is $O(n)$.