Average Case and Amortized Case Analysis

Amortized analysis refers to determining the time-averaged running time for a sequence of operations. It is different from what is commonly referred to as average case analysis, because amortized analysis does not make any assumption about the distribution of the data values, whereas average case analysis assumes the data are not "bad" (e.g., some sorting algorithms do well on "average" over all input orderings but very badly on certain input orderings). That is, amortized analysis is a worst case analysis, but for a sequence of operations, rather than for individual operations. It uses the fact that we are analyzing a sequence to "spread out" the costs (think of insurance where everyone can pay a relatively modest amount despite some catastrophic costs).

While certain operations for a given algorithm may have a significant cost in resources, other operations may not be as costly. Amortized analysis considers both the costly and less costly operations together over the whole series of operations of the algorithm. This may include accounting for different types of input, length of the input, and other factors that affect its performance.

The motivation for amortized analysis is to better understand the running time of certain techniques, where standard worst case analysis provides an overly pessimistic bound. Amortized analysis generally applies to a method that consists of a sequence of operations, where the vast majority of the operations are cheap, but some of the operations are expensive. If we can show that the expensive operations are particularly rare we can "charge them" to the cheap operations, and only bound the cheap operations.
The general approach is to assign an artificial cost to each operation in the sequence, such that the total of the artificial costs for the sequence of operations bounds total of the real costs for the sequence. This artificial cost is called the amortized cost of an operation. In order to analyze the running time, the amortized cost thus is a correct way of understanding the overall running time — but note that particular operations can still take longer so it is not a way of bounding the running time of any individual operation in the sequence.

Average case Running Time: The expected behavior when the input is randomly drawn from a given distribution. The average-case running time of an algorithm is an estimate of the running time for an "average" input. Computation of average-case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences. Often it is assumed that all inputs of a given size are equally likely.

Amortized Running Time: Here the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

Amortized analysis requires knowledge of which series of operations are possible. This is most commonly the case with data structures, which have state that persists between operations. The

basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus "amortizing" its cost.

We may consider two methods for performing amortized analysis: the *aggregate method*, and the *potential method*.

(A) Aggregate analysis determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the amortized cost to be $T(n)/n$.

Example:
Consider a dynamic array that grows in size as more elements are added to it such as ArrayList in Java. If we started out with a dynamic array of size 4, it would take constant time to push four elements onto it. Yet pushing a fifth element onto that array would take longer as the array would have to create a new array of double the current size (8), copy the old elements onto the new array, and then add the new element. The next three push operations would similarly take constant time, and then the subsequent addition would require another slow doubling of the array size.

In general if we consider an arbitrary number of (n + 1) pushes to an array of size n, we notice that push operations take constant time except for the last one which takes time to perform the size doubling operation. Since there were n + 1 operations in total, we can take the average of this and find that inserting elements on to the dynamic array takes

O(n)/(n+1) = O(1)

which is constant time.

(B) In the potential method, a function Φ is chosen that maps states of the data structure to non-negative numbers. If $\boldsymbol{S}$ is a state of the data structure, $\Phi(S)$ may be thought of intuitively as an amount of potential energy stored in that state, alternatively, $\Phi(S)$ may be thought of as representing the amount of disorder in state S or its distance from an ideal state. It represents work that has been accounted for ("paid for") in the amortized analysis, but not yet performed.

The potential value prior to the operation of initializing a data structure is defined to be zero.

Let o be any individual operation within a sequence of operations on some data structure, with $\boldsymbol{S}_{\mathbf{before}}$ denoting the state of the data structure prior to operation o and $\boldsymbol{S}_{\mathbf{after}}$ denoting its state after operation o has completed. Then, once Φ has been chosen, the amortized time for operation o is defined to be

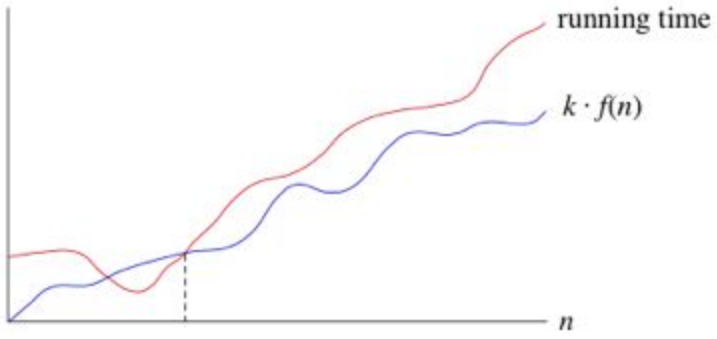
$$T_{amortized}(o) = T_{actual}(o) + C * (\Phi(S_{after}) - \Phi(S_{before}))$$

where C is a non-negative constant of proportionality (in units of time) that must remain fixed throughout the analysis. That is, the amortized time is defined to be the actual time taken by the operation plus C times the difference in potential caused by the operation.

Big Omega (Big-Ω) Notation

Sometimes, we want to say that an algorithm takes *at least* a certain amount of time, without providing an upper bound. We use big-Ω notation.

If a running time is Ω(f(n)), then for large enough n, the running time is at least k · f(n) for some constant k. Here's how to think of a running time that is Ω(f(n)):



We say that the running time is "big-Ω of f(n). We use big-Ω notation for **asymptotic lower bounds**, since it bounds the growth of the running time from below for large enough input sizes.

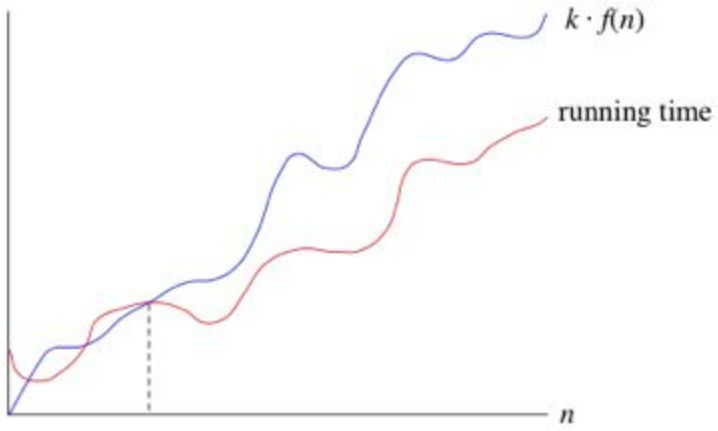
Ω(f(n)) = { g(n) : forall n > n_0. g(n) > f(n) }
In other words, Ω (f(n)) is the set if all functions ***g*** that attain larger values than ***f*** beyond some n.

Thus, Ω (f(n)) defines a lower bound on the time complexity.

Big-O Notation

Although the worst-case running time of binary search is Θ(lg n), it would be incorrect to say that binary search runs in Θ(lg n) time in *all* cases. What if we find the target value upon the first guess? Then it runs in Θ(1) time. The running time of binary search is ***never worse than*** Θ(lg n), but *it's sometimes better*.
It would be convenient to have a form of asymptotic notation that means "the running time grows at most this much, but it could grow more slowly." We use "big-O" notation for just such occasions. If a running time is O(f(n)) then for large enough n, the running time is at most k · f(n) for some constant k.

Here's how to think of a running time that is `O(f(n))`:



We say that the running time is "`big-O of f(n)`" or just "`O of f(n)`." We use `big-O` notation for **asymptotic upper bounds**, since it bounds the growth of the running time from above for large enough input sizes.

O`(f(n)) = { g(n): forall n > n`$_0$ `exists k such that (k. f(n)) > g(n) }`

In other words, **O**(f(n)) is the set if all functions ***g*** that attain smaller values than ***f*** beyond some `n`.

Thus, O(f(n)) defines an upper bound on the time complexity.

Solving Recurrence Equations

A recurrence is a recursive description of a function, or in other words, a description of a function in terms of itself. Like all recursive structures, a recurrence consists of one or more base cases and one or more recursive cases. Each of these cases is an equation or inequality, with some function value f(n) on the left side. The base cases give explicit values for a (typically finite, typically small) subset of the possible values of n. The recursive cases relate the function value f(n) to function value f(k) for one or more integers k < n; typically, each recursive case applies to an infinite number of possible values of n.

Solve the following recurrence relation using **Substitution** method

```
T(n) = T(n-1) + n
```

Note the interesting pattern. The **n^{th}** value is defined as the sum of **$(n-1)^{th}$** value and **n** itself. The basic idea is trivial. As the name implies, go on plugging terms for previous terms till you hit the basis case, like so:

```
T(n) = T(n-1) + n
T(n-1) = T(n-2) + (n-1)
T(n-2) = T(n-3) + (n-2) + (n-1)
T(n-3) = T(n-4) + (n-3) + (n-2) + (n-1)
...
```

You get the idea.

Let's rewrite the same in a different way.

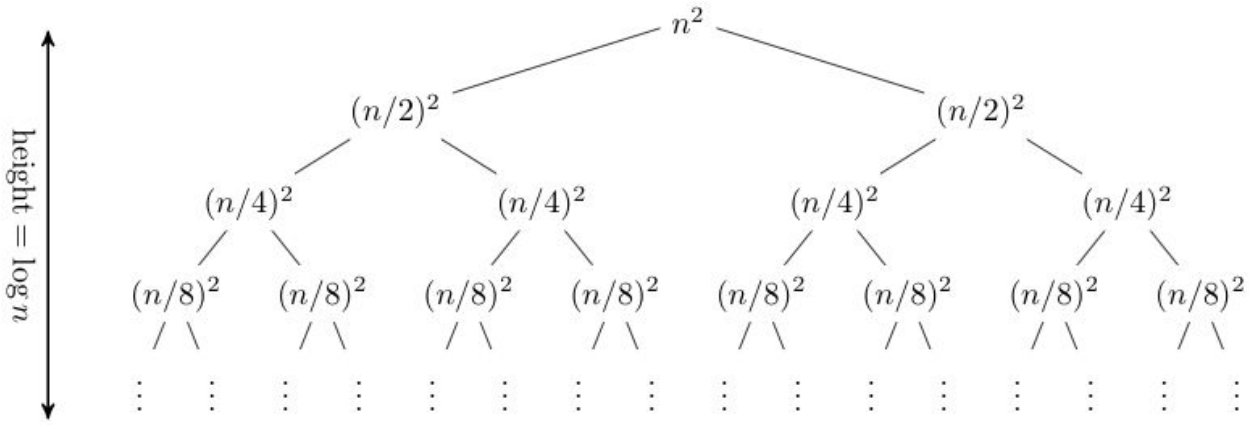
```
T(n) = T(n-1) + n
     = T(n-2) + (n-1) + n
     = T(n-3) + (n-2) + (n-1) + n
     = T(n-4) + (n-3) + (n-2) + (n-1) + n
     = ...
     = T(1) + 2 + 3 + ... + n
     = T(0) + 1 + 2 + 3 + ... + n
     = T(0) + (n * (n+1)/2)
     = O(n^2)
```

Solve the following using **Recursion Tree** method

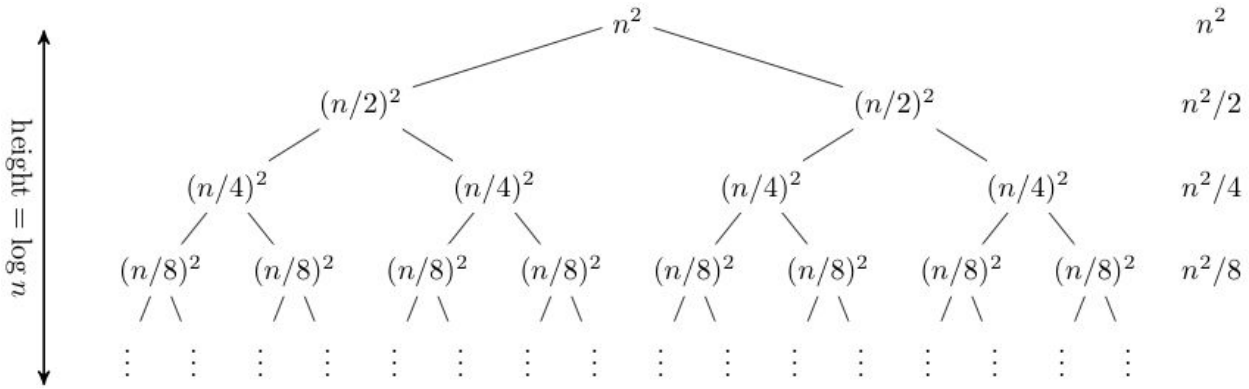
$T(n) = 2T(n/2) + n^2$

A *recursion tree* is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

A little thought shows that the recursion tree for the given recurrence looks like this:



In this case, it is straightforward to sum across each row of the tree to obtain the total work done at a given level:

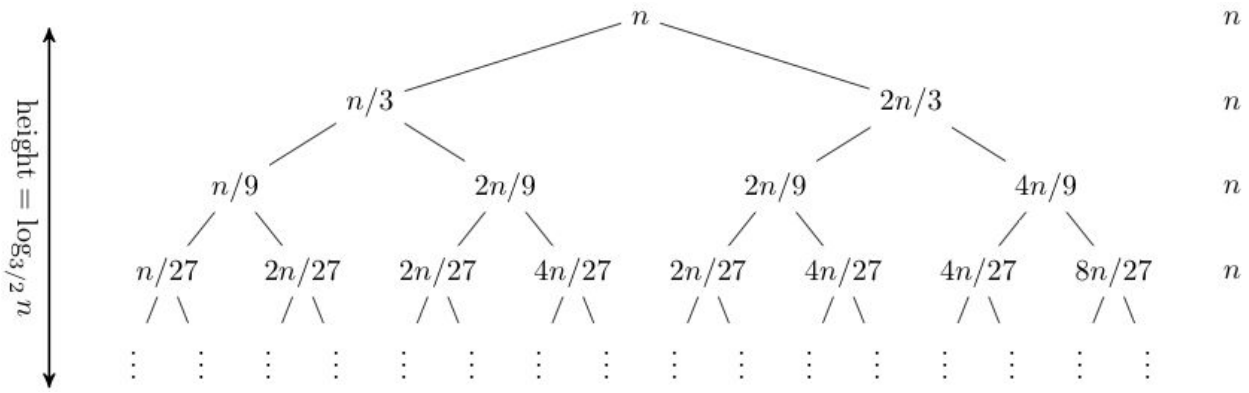


This a geometric series, thus in the limit the sum is $O(n^2)$. The depth of the tree in this case does not really matter; the amount of work at each level is decreasing so quickly that the total is only a constant factor more than the root.

Consider another recurrence equation

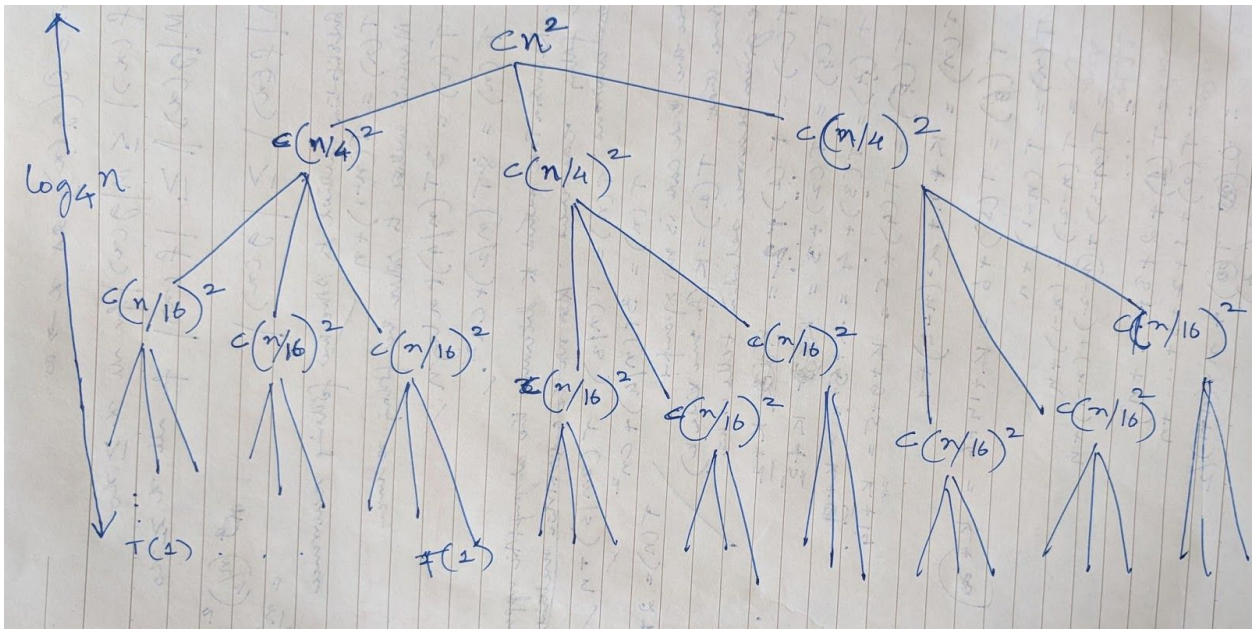
`T(n) = T(n/3) + T(2n/3) + n`

We can visualize the terms in successive stages as shown below:



Note that the tree here is not balanced: the longest path is the rightmost one, and its length is $log_{3/2} n$. Hence our guess for the closed form of this recurrence is *O(n log n)*.

Consider another recurrence relation: $T(n) = 3T(n/4) + cn^2$
The recursion tree could be viewed as shown below:



The analysis of this relation is involved. The analysis yields us an upper bound of **$O(n^2)$**

Master Method

Please see section 4.5 in the “Introduction to Algorithms” book in the data structures github repo. I have also shared this book’s pdf during my lectures. In the textbook (or pdf) pages 95 and 96 show solution to a few recurrences using the master method.

Please refer to the following carefully among them:

(1) $T(n) = 3T(n/4) + n \log n$
(2) $T(n) = 8T(n/2) + cn^2$
(3) $T(n) = T(2n/3) + 1$