Average Case Analysis

October 11, 2011
Worst-case analysis

Worst-case analysis gives an upper bound for the running time of a single execution of an algorithm with a worst-case input and worst-case random choices.

Does this reflect the performance of the algorithm in practice?
Practical considerations

We are often more interested in having a *reasonable estimate* or *reasonable guarantee* of the performance of the algorithm in practice.

- Typical case performance?
- Performance most of the time?
- Performance in all but some pathological cases?
Amortized analysis

Amortized analysis is often used for sequences of operations that update a data structure. It gives an upper bound for the average running time of a single operation in a worst-case sequence of operations, starting from some neutral state.

The fundamental assumption is that while some operations can be expensive, such operations are rare in long sequences of operations.
Typical scenario in amortized analysis

A data structure has space for up to \( n \) elements. Inserting or deleting a single element takes \( \Theta(1) \) time. When the data structure becomes full, the elements are moved into a data structure of size \( 2n \) in \( \Theta(n) \) time. If the structure becomes 1/4 full, the elements are moved into a data structure of size \( n/2 \) in \( \Theta(n) \) time.

In a neutral state, the data structure has \( n/2 \) elements. By defining the potential of a data structure with \( m \) elements as \( \left| m - \frac{n}{2} \right| \), we can easily show that the amortized time for inserting or deleting an element is \( \Theta(1) \).
Average case analysis gives an *upper bound* for the *expected running time* of a *single execution* of a *deterministic algorithm* with a *random input* selected according to some *distribution*.

Is the average case also typical case?
Average case analysis of Quicksort

**function** Quicksort(A):

  if |A| ≤ 1:
    **return** A

  Select *pivot p* from A.

  *Partition A* into smaller, equal, and larger than *p*.

  **return** Quicksort(smaller) + equal + Quicksort(larger)

*Quicksort* is an interesting algorithm for average case analysis, because its worst-case complexity is $\Theta(n^2)$, while the typical performance is much better than for other $\Theta(n^2)$-time algorithms (e.g. *Insertion sort*).
Average case analysis of Quicksort

In the analysis, we always choose the first element as the pivot. The input is assumed to be a permutation of integers \{1, \ldots, n\}, with each of the permutations equally likely.

The running time of QuickSort is determined by the number of element comparisons, plus \( \Theta(n) \) to cover the base cases. Integers \( i \) and \( j \) are compared at most once. This happens, if one of \( i \) and \( j \) is chosen as a pivot before any of \( i + 1, \ldots, j - 1 \) is chosen. This happens with probability \( 2/(j - i + 1) \).

This suggests that we could use indicator variables in the analysis.
Average case analysis of Quicksort

Let $X_{i,j}$ indicate whether integers $i$ and $j$ (with $i < j$) are compared during sorting. Now $X_{i,j} = 1$ with probability $2/(j - i + 1)$, and $X_{i,j} = 0$ otherwise.

The analysis is greatly simplified by the linearity of expectations:

$$E \left[ \sum_{1 \leq i < j \leq n} X_{i,j} \right] = \sum_{1 \leq i < j \leq n} E[X_{i,j}].$$
Average case analysis of Quicksort

\[
E \left[ \sum_{1 \leq i < j \leq n} X_{i,j} \right] = \sum_{1 \leq i < j \leq n} E[X_{i,j}] = \sum_{1 \leq i < j \leq n} \frac{2}{j - i + 1} \\
= 2 \cdot \sum_{i=1}^{n-1} \sum_{j=2}^{n-i+1} \frac{1}{j} \leq 2 \cdot \sum_{i=1}^{n-1} H_i \\
= \Theta(n \log n),
\]

where \( H_i = \Theta(\log i) \) is the \( i \)th harmonic number.

Hence the expected running time of Quicksort is \( \Theta(n \log n + n) = \Theta(n \log n) \).
Average case analysis of Quicksort

We can also base the average case analysis on the following recurrence:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n \leq 1 \\
\frac{1}{n} \sum_{i=1}^{n} (T(i - 1) + T(n - i) + \Theta(n)) & \text{otherwise}
\end{cases}
\]

The general case is further divided into cases, where the pivot is the \( i \)th smallest element in the array.
Average case analysis of Quicksort

We start by simplifying the general case:

\[
T(n) = \frac{1}{n} \sum_{i=1}^{n} \left( T(i - 1) + T(n - i) + \Theta(n) \right)
\]

\[
= \frac{2}{n} \sum_{i=0}^{n-1} T(i) + \Theta(n).
\]

For large enough \( n \), it holds that

\[
nT(n) - (n - 1)T(n - 1) = 2T(n - 1) + \Theta(n).
\]
Average case analysis of Quicksort

\[
T(n) = \frac{n+1}{n} T(n-1) + \Theta(1)
\]
\[
= \frac{n+1}{n-1} T(n-2) + \Theta \left( 1 + \frac{n+1}{n} \right)
\]
\[
= \cdots
\]
\[
= \frac{n+1}{2} T(1) + \Theta \left( (n+1) \sum_{i=3}^{n+1} \frac{1}{i} \right)
\]
\[
= \Theta(n \log n).
\]
Average case analysis of Quicksort

A typical implementation uses Quicksort for arrays larger than \( k \) elements, and sorts small arrays with Insertion sort. As Insertion sort can move an element for at most \( k - 1 \) positions, the total time complexity for all Insertion sorts is \( \Theta(nk) \).

By setting \( T(n) = \Theta(1) \) for \( n \leq k \), we get

\[
T(n) = \frac{n + 1}{k + 1} T(k) + \Theta \left( (n + 1) \sum_{i=k+2}^{n+1} \frac{1}{i} \right) = \Theta \left( n \log \frac{n}{k} \right)
\]

for the Quicksort phase, for a total of \( \Theta(n(k + \log(n/k))) \).
Average case analysis

Average case analysis gives an upper bound for the expected running time of a single execution of a deterministic algorithm with a random input selected according to some distribution.

OR

Average case analysis gives an upper bound for the expected running time of a single execution of a randomized algorithm with a worst-case input.
Average case analysis

Randomized Quicksort selects a random element as the pivot. The same average case analysis works for this variant as well.

Should we draw the same conclusions from the analysis of deterministic Quicksort and randomized Quicksort?
Average case analysis

For randomized Quicksort, *any input* may require $\Omega(n^2)$ time, but this is *extremely unlikely*.

For deterministic Quicksort, *some inputs* require $\Omega(n^2)$ time. If the first element is always chosen as the pivot, an array that is *already sorted* is one of these inputs.

It is hard to find an input distribution that reflects the *actual distribution* of inputs in practice, and is *easy to analyze*. 
Average case analysis

Given a distribution of running times,

- *worst-case analysis* gives an upper bound for the maximum, while
- *average case analysis* gives an upper bound for the expected value.

In statistics, there is no single parameter that always captures the relevant properties of a distribution. Similarly, there is no single way of analyzing an algorithm that always gives reasonable bounds for its performance in practice. One has to understand the nature of a particular algorithm to know, which method of analysis accurately describes its performance.
Smoothed analysis gives an upper bound for the expected running time of a single execution of an algorithm for a random input that is close to the worst-case input.


Perhaps the most important advance in the analysis of algorithms in the 2000s.
Discrete Optimization (4 cr), period II, Mon, Wed 14–16. Registration starts today!

The canonical form of a linear optimization problem is

\[
\begin{align*}
\text{maximize} & \quad c^T x \\
\text{subject to} & \quad Ax \leq b \\
\text{and} & \quad x \geq 0.
\end{align*}
\]

With \( n \) variables and \( m \) constraints, a solution \( x \) is a point in \( \mathbb{R}^n \), and the \( m \) half-spaces defined by \( A \) and \( b \) limit the feasible region in \( \mathbb{R}^n \). The goal is to maximize the linear real-valued target function \( c^T x \) in the feasible region.
Simplex algorithm (simplified)

function Simplex(A, b, c):
    Find a feasible solution $x$.
    repeat:
        Find a direction that increases $c^T x$.
        Move $x$ in that direction as far as possible.
    until the solution no longer improves.
    return $x$

The average running time of one version of Simplex is $O(\sqrt{n})$, yet the algorithm is very efficient in practice.
Spielman and Teng add Gaussian noise with normalized standard deviation $\sigma$ to the constraints $A$ and $b$. They then proceed to show that the running time of the Simplex algorithm is polynomial in $n$, $m$, and $1/\sigma$.

Even though the average running time is exponential, most of it is produced by small neighborhoods of hard instances. If the constraints are imprecise (e.g. due to measurement errors), then these hard instances are almost never found in practice.
String Processing Algorithms (4 cr)

Period II, Tue, Thu 12–14
Juha Kärkkäinen

- Exact and approximate string matching.
- String sorting and suffix sorting.
- Suffix trees, suffix arrays, and related data structures.
- Data structures for sets of strings.

Period III: Data Compression Techniques (4 cr), Project in String Processing Algorithms (2 cr)

A major research area at CS Department.