Lesson 3
Average Case Analysis

Assessing Performance Through Analysis And Synthesis

Algorithms

Wholeness of the Lesson

Average-case analysis of performance of an algorithm provides a measure of the typical running time of an algorithm. Although computation of average-case performance generally requires deeper mathematics than computation of worst-case performance, it often provides more useful information about the algorithm, especially when worst case analysis yields exaggerated estimates. Likewise, as discussed in SCI, more successful action results from a deeper dive into silence, into pure intelligence, just as, in archery, the arrow flies truer and hits its mark more consistently if it is pulled back farther on the bow.

Overview

(1) Introducing average-case analysis: Simple sorting algorithms

(2) A second example of average-case analysis: Searching a list

(3) A simple probability model: The coin-flipping experiment

Analysis of Simple Sorting Algorithms

- BubbleSort, SelectionSort and InsertionSort are among the simplest sorting methods and admit straightforward analysis of running time. For each we will consider best case and worst case running times.
- We will find that InsertionSort has a best case running time of O(n) but a worst-case running time of $\Theta(n^2)$. A natural question is: Which of these running times is *typical*? *On average*, how does InsertionSort perform?
- For some algorithms, the worst-case running time does not give an accurate picture, since worst-cases may happen very rarely. In those cases, some kind of average-case analysis is preferable. In practice there are many ways to accomplish this. We look at one approach to analyze sorting algorithms like InsertionSort, and then develop some simple tools for handling other kinds of algorithms.

Analysis of BubbleSort.

```
void sort(){
    int len = arr.length;
    for(int i = 0; i < len; ++i) {
      for(int j = 0; j < len−1; ++j) {
        if(arr[j] > arr[j+1]){
          swap(j,j+1);
        }
      }
    }
}
void swap(int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

Correctness of BubbleSort.

Loop Invariant. For $0 \leq i \leq n-1$,

$$I(i): \texttt{arr[n-i-1]..arr[n-1]} \text{ are in final sorted order}$$

If Loop Invariant holds for $0 \leq i \leq n-1$, it implies BubbleSort produces a sorted array.

Proof that Loop Invariant holds. Certainly $I(0)$ is true (when i=0 pass completes, largest element has been placed at the end). Assume $I(i)$ holds; we show `arr[n-i-2]..arr[n-1]` are in final sorted order. As inner loop runs, the largest element `max` among the elements `arr[0], arr[1],..., arr[n-i-2]` is pushed to the right as far as possible. Since the elements `arr[n-i-1]` through `arr[n-1]` are in final sorted order, `max` is not pushed into any of those final slots. Therefore, it is placed in slot `n-i-2`.

A. *"Every Case" Analysis.*

- Because there are two loops, nested, both depending on n it is $\Theta(n^2)$
- In this implementation, there is no "best" or "worst" case.

B. *Possible Improvements.*

- It is possible to implement BubbleSort slightly differently so that in the best case (which means here that the input is already sorted), the algorithm runs in $\Theta(n)$ time. (Exercise)
- As the Loop Invariant shows, at the end of iteration i, the values in `arr[n-i-1]` through `arr[n-1]` are in final sorted order. This observation can be used to shorten the inner loop. The result is to cut the running time in half (though it must still be $\Theta(n^2)$). (Exercise)

Analysis of SelectionSort.

```
void sort(){
    int len = arr.length;
    int temp = 0;
    for(int i = 0; i < len; ++i) {
      int nextMinPos = minpos(i,len-1);
      swap(i,nextMinPos);
    }
}
void swap(int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
int minpos(int bottom, int top){
    int m = arr[bottom];
    int index = bottom;
    for(int i = bottom+1; i <= top; ++i) {
      if(arr[i]<m){
        m=arr[i];
        index=i;
      }
    }
    return index;
}
```

Optional: Correctness of SelectionSort.

Loop Invariant. For $0 \leq i \leq n-1$,

$$I(i): \texttt{arr[0]..arr[i]} \text{ are in final sorted order}$$

If Loop Invariant holds for $0 \leq i \leq n-1$, it implies SelectionSort produces a sorted array.

Proof that Loop Invariant holds. At the end of the $i = 0$ pass, the minimum element of the array has been placed in position 0, so $I(0)$ holds. Assuming `arr[0]..arr[i]` are in final sorted order, as inner loop runs, the position `pos` of the smallest element among the elements `arr[i+1]`, `arr[i+2],...,arr[n-1]` is obtained and `arr[pos]`, `arr[i+1]` are swapped. Since `arr[0], arr[1], . . .,arr[i]` are already in sorted order, iteration $i+1$ results in `arr[0]..arr[i], arr[i+1]` being in final sorted order.

A. *"Every-Case" Analysis.*

- Because there are two loops, nested, depending on n, it is $\Theta(n^2)$.
- There is no "best case" or "worst case" for SelectionSort.

Analysis of InsertionSort

```
void sort(){
    int len = arr.length;
    int temp = 0;
    int j = 0;
    for(int i = 1; i < len; ++i) {
      temp = arr[i];
      j=i;
      while(j>0 && temp < arr[j-1]){
        arr[j] = arr[j-1];
        j--;
      }
      arr[j]=temp;
    }
}
```

Optional: Correctness of InsertionSort.

Loop Invariant. For $0 \leq i \leq n-1$,

$$I(i): \texttt{arr[0]..arr[i]} \text{ are in sorted order}$$

Note in this case, the invariant does not say `arr[0]..arr[i]` are in *final* sorted order. If Loop Invariant holds for $0 \leq i \leq n-1$, it implies Insertion produces a sorted array.

Proof that Loop Invariant holds. $I(0)$ holds since `arr[0]` is just a single element (so it is automatically sorted). Assume $I(i)$ is true. As the $i+1$ inner loop runs, `arr[i+1]` is compared with each value x in `arr[i], arr[i-1]`... until an x is found for which `arr[i+1]` $\not< x$. `arr[i+1]` is then placed to the right of x. This means `arr[i+1]` has a value between x and the next larger value. It follows that `arr[0]..arr[i], arr[i+1]` are now arranged in sorted order.

A. *Best-Case Analysis.* The best case for InsertionSort occurs when the input array is already sorted. In this case, the condition in the inner while loop always fails, so the code inside the loop never executes. The result is that execution time inside each outer loop is constant, and so running time is $O(n)$.

B. *Worst-Case Analysis.* Since there are two loops, nested, even in the worst case, the running time is only $\Theta(n^2)$. The worst case for InsertionSort occurs when the input array is reverse-sorted. In this case, in pass $\#i$ of the outer for loop the inner while loop must execute all its statements i times approximately, and so execution time is proportional to $1 + 2 + \ldots + n - 1 = \Theta(n^2)$. Therefore, worst-case running time is $\Theta(n^2)$.

C. *Average-Case Analysis.* It is reasonable to expect that typically, the inner while loop will not work as hard as it does in the worst-case. The result of average case analysis here actually applies to many simple sorting algorithms.

Inversion-Bound Sorting Algorithms

A. In an array `arr` of integers, an *inversion* is a pair $(\texttt{arr}[i], \texttt{arr}[j])$ for which $i < j$ and $\texttt{arr}[i] > \texttt{arr}[j]$.

Example. The array $\texttt{arr} = \{34, 8, 64, 51, 32, 21\}$ has nine inversions:

$$(34,8), (34, 32), (34, 21), (64, 51), (64, 32),$$
$$(64, 21), (51, 32), (51, 21), (32, 21).$$

B. **Theorem** (*Number of Inversions Theorem*). Assuming that input arrays contain no duplicates and values are randomly generated, the expected number of inversions in an array of size n is $\frac{n(n-1)}{4}$.

Proof. Given a list L of n distinct integers, consider L_r, obtained by listing L in reverse order. Suppose x, y are elements of L and $x < y$. These elements must occur in inverted order in either L or L_r (but not both). Therefore every one of the $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of elements from L occurs as an inversion exactly once in L or L_r. Therefore, on average, one-half of these inversions occur in L itself.

NOTE: By "expected number of inversions" we mean the number of inversions *on average*. This is the same idea as when we say "If you flip a coin 100 times, *on average*, heads will turn up 50 times."

C. **Corollary**. Suppose a sorting algorithm always performs at least as many comparisons as there are inversions on any input array `arr`. Then the average-case running time of this algorithm acting on arrays of distinct elements is $\Omega(n^2)$.

D. **Definition.** A sorting algorithm that always performs at least as many comparisons as there are inversions on any input array `arr` is called an *inversion-bound* algorithm.

Observations About Inversion-Bound Algorithms

We assume all elements of input arrays are distinct — average case analysis does not address the case in which the array has duplicates.

A. BubbleSort, SelectionSort, and InsertionSort can all be shown to be inversion-bound.

B. **Proof that InsertionSort Is Inversion-Bound**. Suppose `arr` is an input array, $i < j <$ `arr.length` and $x =$ `arr`$[i] >$ `arr`$[j] = y$. At some point the initial part of the array consisting of all values $< y$ will be in sorted order, and y will need to be placed. y will still be to the right of x. So, in the loop that does the placing, y will have to be compared with x.

Comparing Performance of Simple Sorting Algorithms

A. Demos give empirical data for comparison. (Demos)

B. *Swaps are expensive.* Notice swaps involve roughly seven primitive operations. This is more significant than copies and comparisons. BubbleSort performs (on average)
$\Theta(n^2)$ swaps whereas SelectionSort and InsertionSort perform only $O(n)$ swaps (a "swap" for InsertionSort begins when an element is placed in `temp` and ends when it is placed in its final position). This difference explains why BubbleSort is so much slower than the other two. (Empirical studies show BubbleSort is 5 times slower than InsertionSort and 40% slower than SelectionSort.)

A Refinement of InsertionSort: LibrarySort

Bender, Farach-Colton, Mosteiro observed that InsertionSort is slower than necessary for two reasons:

1. In the ith iteration, the search for where to place the next element among the first (already sorted) i elements is not optimized (*binary search* could be used instead)
2. When the correct place for the next element has been found, the effort to shift all larger elements to the right is slower than necessary (library analogy: leave gaps to make room for new additions)

They implement their ideas for optimizations in a paper that introduces *LibrarySort*. (Their paper is entitled *INSERTION SORT is* $\mathrm{O}(n \log n)$.) Their algorithm achieves average case running time of $\mathrm{O}(n \log n)$. Demo of LibrarySort

Main Point

A sorting algorithm is *inversion-bound* if it requires, on any given input array, at least as many comparisons as there are inversions in the array. Inversion-bound sorting algorithms always have an asymptotic running time that is $\Theta(n^2)$. Selection Sort, Insertion Sort and Bubble Sort are examples of inversion-bound sorting algorithms.

Maharishi explains in SCI that knowledge is different in different states of consciousness. When consciousness is bounded, it is simply not possible to see higher possibilities in life. When consciousness expands, more possibilities are seen; new directions can unfold; old problems can be solved in new ways.

Overview

(1) $\surd$ Introducing average-case analysis: Simple sorting algorithms

(2) A second example of average-case analysis: Searching a list

(3) A simple probability model: The coin-flipping experiment

Average-Case Analysis: Searching a List

- Example of average case analysis: What is the average case running time to find an element in an array of length n?

  Solution:

  (A) Intuitively, if the element is present in the array, we need to run through only half the array, on average. Therefore, average-case running time is proportional to $n/2$, since examining each slot in the array costs just O(1). Notice that under worst-case analysis, running time is n (since desired element may occur for the first time in the last slot, or not at all).

  (B) Terminology: We say that the *expected number of slots that need to be checked is* $n/2$.

  (C) We arrived at $n/2$ using intuitive reasoning. We can compute the correct value more precisely by asking: What is the average slot number in an array (slot numbers are the array indices: $0, 1, 2, ..., n-1$)?

$$\begin{aligned}\text{average slot number} &= \frac{0+1+2+\cdots+(n-1)}{n} \\ &= \frac{[(n-1)\cdot n]/2}{n} \\ &= \frac{n-1}{2}\end{aligned}$$

(D) We can rewrite this computation in the following way:

$$\frac{0+1+2+\cdots+(n-1)}{n} = \left(\frac{1}{n}\cdot 0\right) + \left(\frac{1}{n}\cdot 1\right) + \ldots + \left(\frac{1}{n}\cdot (n-1)\right)$$

Since it is equally likely to find the desired element in any of the slots, the probability of finding the desired element in slot 0 is $\frac{1}{n}$; in slot 1 is $\frac{1}{n}$; etc.

So our formula says: Multiply each slot number by the probability of finding the element in that slot.

Formula for Computing Expected Value.
Suppose performing an experiment could result in any of n integer outcomes, $s_1, \ldots, s_n$. And suppose the probability of the outcome s_i is p_i. Then the *expected value* of the experiment is

$$p_1 \cdot s_1 + p_2 \cdot s_2 + \ldots + p_n \cdot s_n.$$

In the example of finding the expected slot number, we had $p_1 = p_2 = \ldots = p_{n-1} = \frac{1}{6}$ and $s_1 = 0, s_2 = 1, \ldots, s_n = n - 1$.

Overview

(1) $\surd$ Introducing average-case analysis: Simple sorting algorithms

(2) $\surd$ A second example of average-case analysis: Searching a list

(3) A simple probability model: The coin-flipping experiment

The Coin Flipping Experiment

Problem. Repeatedly flip a coin. How many flips are required to get "heads"?

In the worst case, what is the answer?

Problem. Repeatedly flip a coin. How many flips are required to get "heads"?

In the worst case, what is the answer? Answer: In the worst case, "heads" never comes up.

We can see that in this case, worst case analysis is not useful. Our next question will be: What is the expected number of flips required to get heads?

Solution.

- This is an example of computed expected value using our formula

  $$p_1 \cdot s_1 + p_2 \cdot s_2 + \ldots + p_n \cdot s_n,$$

  where for each i, s_i is an outcome and p_i is the probability that this outcome occurs. [In this case, however, we will need to consider *infinitely many* different possible outcomes.]
- Here the outcomes are the different numbers of flips required:

  $$1, 2, 3, 4, \ldots$$
- Probability that "heads" occurs in just one flip is $\frac{1}{2}$
- Probability that "heads" occurs for the first time on the second flip means that on the first flip, we got "tails". So what is the probability of getting "TF" (first "tails" then "heads")? It is $\frac{1}{4}$.
- We can arrive at $\frac{1}{4}$ another way: What is the probability of getting tails on the first flip and head on the second flip? Probability of tails on the first flip is $\frac{1}{2}$; probability of heads on the second flip is also $\frac{1}{2}$. Since these events are *indepedendent* (whether one happens does not influence whether the other happens), probability theory says that the probability of both happening is found by multiplying: $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$.

◦ Similar reasoning shows:

$$\begin{aligned}
\text{probability of heads on 1st flip} &= \frac{1}{2} \\
\text{probability of heads on 2nd flip} &= \frac{1}{4} \\
\text{probability of heads on 3rd flip} &= \frac{1}{8} \\
\text{probability of heads on 4th flip} &= \frac{1}{16} \\
\text{probability of heads on 5th flip} &= \frac{1}{32} \\
&\vdots \quad\quad \vdots
\end{aligned}$$

◦ These computations lead to the following:

$$\begin{aligned}
&\text{Expected number of flips} \\
&\quad = \left(\frac{1}{2} \cdot 1\right) + \left(\frac{1}{4} \cdot 2\right) + \left(\frac{1}{8} \cdot 3\right) + \cdots \\
&\quad = \sum_{i=1}^{\infty} \frac{i}{2^i} \\
&\quad = 2.
\end{aligned}$$

Optional: Wait, Where Did "2" Come From?

- Observation: Define $g(x)$ by

$$g(x) = \sum_{k=0}^{\infty} ax^k$$

Whenever $|x| < 1$, this sum is equal to

$$\frac{a}{1-x}.$$

Since the series is absolutely convergent for $|x| < 1$, the derivative can be computed by

$$g'(x) = \sum_{k=0}^{\infty} kax^{k-1}. \tag{1}$$

The derivative also equals

$$\frac{d}{dx}\,\frac{a}{1-x} = \frac{a}{(1-x)^2}. \tag{2}$$

The expression

$$\sum_{i=1}^{\infty} i \cdot \left(\frac{1}{2}\right)^i = \left(\frac{1}{2}\right) \sum_{i=1}^{\infty} i \cdot \left(\frac{1}{2}\right)^{i-1}$$

from previous slide has the form of (1) (the $i = 0$ case is irrelevant). Therefore by (2)

$$\sum_{i=1}^{\infty} i \cdot \left(\frac{1}{2}\right)^i = \left(\frac{1}{2}\right)\left(\frac{1}{(1-\frac{1}{2})^2}\right) = 2.$$

- Therefore: *The expected number of flips required to get "heads" is 2.*
- Similar reasoning shows that *the expected number of die rolls required to get a 1 is 6.*

Theorem [*Expected number of trials for success*]. Suppose an experiment is performed repeatedly, and the probability of "success" is p (where p is a real number between 0 and 1), and is not influenced by previous successes or failures. Then the expected number of trials to obtain a success is $\frac{1}{p}$.

- Likewise, for any integer $k > 0$, *The expected number of flips required to get exactly k "heads" is $2k$*

This leads to the following Theorem:

Theorem [*Expected number of trials for k successes*]. Suppose k is a positive integer. Suppose an experiment is performed repeatedly, and the probability of "success" is p (where p is a real number between 0 and 1), and is not influenced by previous successes or failures. Then the expected number of trials to obtain k successes is $\frac{k}{p}$.

Overview

(1) $\surd$ Introducing average-case analysis: Simple sorting algorithms

(2) $\surd$ A second example of average-case analysis: Searching a list

(3) $\surd$ A simple probability model: The coin-flipping experiment

Connecting the Parts of Knowledge
To the Wholeness of Knowledge
Inversion-Bound Algorithms

1 Insertion Sort is an inversion-bound algorithm that sorts by examining each successive value x in the input list and searches the already sorted section of the array for the proper location for x.

2 Library Sort is also a sorting algorithm which, like Insertion Sort, proceeds by examining each successive vaule x in the input list and searches the already sorted section of the array for proper placement. However, in this algorithm, spaces are created in the already sorted section in each pass, and searching the already sorted section is done using Binary Search. The result of these refinements is that Library Sort exceeds the limitations of an inversion-bound sorting algorithm and has average case running time that is O($nlogn$).

3 *Transcendental Consciousness* is the field pure intelligence, the home of all knoweldge, that field "by which all else is known."

4 *Impulses within the Transcendental field.* Maharishi explains that knowledge has organizing power; pure knowledge has infinite organizing power.

5 *Wholeness moving within itself.* In Unity Consciousness, the field of unlimited boundlessness is appreciated in each boundary of existence as its true nature, no different from one's own Self.