More Average Case Analysis And Amortized Analysis

Algorithms

Corazza

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, worst case and average case running times.

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;
}
```

A. *“Every Case” Analysis.*

- Because there are two loops, nested, it is $O(n^2)$ – but could it be faster than n^2 (maybe $O(n \log n)$?).
- Each element in the array is compared with all the other elements ($n-1$ comparisons) n times. So the algorithm is $\Omega(n^2)$.
- Therefore, BubbleSort runs in $\Theta(n^2)$ even in the best 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 $O(n)$ time. (Exercise)

- Notice that at the end of the first pass through the outer loop, the largest element of the array is in its final sorted position. After the next pass, the next largest element is in its final sorted position. After the ith pass ($i = 0, 1, 2, \ldots$), the largest, second largest,. . ., $i + 1$st largest elements are all in their final sorted positions. 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 $\Omega(n^2)$ – see below). (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;
}
```

A. *"Every-Case" Analysis.*

- Because there are two loops, nested, it is $\mathrm{O}(n^2)$ – but could it be faster than n^2 (maybe $\mathrm{O}(n \log n)$?).
- During pass #i, the minpos function must locate the minimum value in the range $i..n-1$, which requires $\Theta(n-i-1)$ comparisons. Therefore, at least $1+2+\ldots+n-1 = \Omega(n^2)$ steps are required.
- Therefore, SelectionSort runs in $\Theta(n^2)$.

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;
    }
}
```

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 $\mathrm{O}(n)$.

B. *Worst-Case Analysis.* Since there are two loops, nested, even in the worst case, the running time is only $\mathrm{O}(n^2)$. 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 $i - 1$ times, and so execution time is proportional to $1 + 2 + \ldots + n - 1 = \Omega(n^2)$. 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.

For a given array `arr` of distinct integers, let $Inv_{\texttt{arr}}$ denote the number of inversions that occur in `arr`.

C. **Corollary**. Suppose a sorting algorithm always performs at least $Inv_{\texttt{arr}}$ comparisons 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 $Inv_{\texttt{arr}}$ comparisons 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 are all inversion-bound.

- *BubbleSort*. Suppose `arr` is an input array, $i < j <$ `arr.length` and $x =$ `arr`$[i] >$ `arr`$[j] = y$. BubbleSort always compares adjacent elements. During BubbleSort, wherever x and y may be in the array, they will eventually become adjacent and then will be compared:

  i All elements between the ith and the jth that are bigger than $x =$ `arr`$[i]$ are pushed to the right of $y =$ `arr`$[j]$ (result: no element between x and y is bigger than x)

  ii x is pushed to the right of all elements between x and y, including y.

  This shows that for every inversion in the input array, at least one (uniquely determined) comparison is performed in BubbleSort.

- *SelectionSort*. Suppose `arr` is an input array, $i < j <$ `arr.length` and $x = \texttt{arr}[i] > \texttt{arr}[j] = y$. At some point during SelectionSort, all elements of the array less than y will have been moved to the front of the array, to the left of x. In the next pass — called the *y pass* — a crucial comparison (called the *crucial comparison with x*) will determine that x is not the min of the remaining elements. (This comparison may or may not be a comparison between x and y.) This gives us a mapping between inversions and unique comparisons: For any inversion (x, y) that occurs in the array, there is a unique crucial comparison of x that occurs in the y-pass.

- *InsertionSort*. 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.

B. **Observation**. The average-case running time of BubbleSort, SelectionSort, and InsertionSort is $\Theta(n^2)$.

Comparing Performance of Simple Sorting Algorithms

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

C. *SelectionSort vs. InsertionSort.* For both algorithms, after the ith iteration, the first i elements are in sorted order. In placing the next element. InsertionSort checks on average half of these first i elements, whereas SelectionSort compares with all elements in the remainder of the array. However, the tail of the array is shrinking, so the lost time at first is made up later. It is not clear from the code that one is faster than the other. However, empirical tests show that, generally, InsertionSort outperforms 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

Amortized Analysis

- Average case analysis is often difficult, sometimes requiring sophisticated probability analysis. A more accessible type of average case analysis, not requiring probability, is *amortized analysis*.
- In amortized analysis, we determine the total running time T_{total} of several algorithms or operations working together, for a total of, say, n executions, and then declare that the amortized running time of any one of the operations is T_{total}/n.
- For each operation considered in an amortized analysis, we use its worst-case running time. Therefore, an amortized analysis *guarantees the average performance of each operation in the worst case.*

- **Definition**. Suppose S is a collection of operations that can be performed on a data structure. For each s in S, let $c(s)$ denote the cost of s, that is, the actual number of primitive operations required to execute s.

  Intuition. We think of $c(s)$ as the cost in “cyberdollars” to run s on a computer that you are renting.

- **Definition**. An *amortized cost function* $\hat{c}$ is also defined on each operation in S.

  Intuition. We think of $\hat{c}(s)$ as the number of cyberdollars we charge a user who wishes to execute operation s on the computer we have rented.

- **Definition**. An amortized cost function $\hat{c}$ on S *bounds* S if, for any sequence of n operations $s_1, s_2, \ldots, s_n$ from S (repetitions allowed) we have, for $1 \leq j \leq n$:

  $$\sum_{i=1}^{j} \hat{c}(s_i) \geq \sum_{i=1}^{j} c(s_i).$$

  The *amortized profit at stage j* is the difference

  $$\sum_{i=1}^{j} \hat{c}(s_i) - \sum_{i=1}^{j} c(s_i).$$

  The *amortized running time* of the sequence $s_1, s_2, \ldots, s_n$ is the sum

  $$\sum_{i=1}^{n} \hat{c}(s_i).$$

  and the *amortized running time of each operation* is the average

  $$\Big(\sum_{i=1}^{n} \hat{c}(s_i)\Big)/n.$$

  NOTE: The first inequality says that a bounding amortized cost function must always yield nonnegative profit.

Motivation. The purpose of creating an amortized cost function is to make it easier to compute the running time of a sequence of operations from S. The actual running time is no worse than the amortized running time. Therefore, the amortized running time for an operation in S represents a type of average-case bound on its running time.

- **Theorem**. Suppose $s_1, s_2, \ldots, s_n$ is a sequence of operations from S having amortized running time T_A. If T is the actual running time of $s_1, s_2, \ldots, s_n$, then

$$T \quad \text{is} \quad \mathrm{O}(T_A).$$

Example: *A Clearable Table*

Start with an empty String array of fixed length N (assume N is big enough for all operations we will perform – resizing will never be necessary). Let S consist of two operations:

$$\texttt{add(x)} = \text{inserts String } \texttt{x} \text{ into next avail slot}$$
$$\texttt{clear()} = \text{replaces all Strings in array with } \texttt{null}\text{s}$$

We wish to determine the average running time required by an arbitrary sequence of n operations from S.

- *First Try.* If we try doing a worst-case analysis in a naive way, we might reason as follows: In the worst case `clear()` occurs at least half the time in the sequence of n operations, and, in the worst case, each `clear()` operation requires $O(n)$ steps. Therefore, the worst-case running time is $O(n^2)$.

The First Try is not incorrect, but not precise either. The worst case described cannot actually happen. An amortized analysis will show that the running time is always $\mathrm{O}(n)$.

- *Second Try: Amortized Analysis.*

  Actual costs

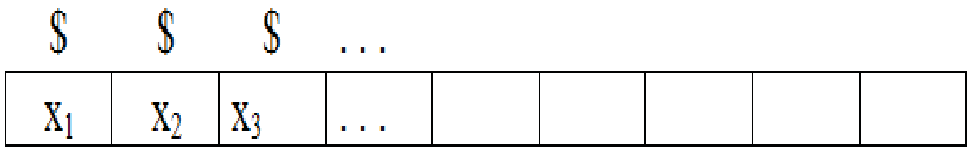
  - $c(\texttt{add}) = 1$ (= 1 cyberdollar)
  - $c(\texttt{clear}) = k$ (= k cyberdollars), where k is # elements currently in array

  Amortized costs

  - $\hat{c}(\texttt{add}) = 2$ (= 2 cyberdollars)
  - $\hat{c}(\texttt{clear}) = 0$ (= 0 cyberdollars)

  NOTE: The cleverness in amortized analysis is in devising the amortized cost function; this requires practice!

Claim. The amortized cost function $\hat{c}$ bounds
$S = \{$`add, clear`$\}$.



Proof. Using `add` to insert a new String in the array always increases current amortized profit by 1 cyberdollar. Therefore, whenever `clear` is called with k elements in the array, there are exactly k cyberdollars of credit available, so that amortized profit remains nonnegative. In other words, for $1 \le j \le n$,

$$\sum_{i=1}^{j} \hat{c}(s_i) \ge \sum_{i=1}^{j} c(s_i).$$

Conclusion. Therefore, the amortized cost of a sequence of n operatons from S is $\mathrm{O}(2n) = \mathrm{O}(n)$, and so the amortized cost of a single operation from S is $\mathrm{O}(n)/n = \mathrm{O}(1)$.

Example: *Resizing An Array.*

The Problem. Suppose you are creating your own ArrayList data structure. In this data structure, you store objects in a background array and implement List operations, such as add, get, and remove, by manipulating the background array. When the background array is full and an add operation is invoked, it is necessary to resize the background array. If the background array currently has n elements, how big should the new background array be?

Here we compare the running times for a sequence of add operations using two different strategies:

1. When resizing is required, double the size of the array
2. When resizing is required, increase the size by a fixed increment.

The Size-Doubling Strategy. Our goal is to show that each add operation has amortized running time of O(1) when the size-doubling strategy is used. We begin with an empty array of length 1 and repeatedly attempt to add new elements; when the array is full, we invoke a resize operation which creates a new array of twice the previous size and copies the old elements into the initial part of the new array.

Operations. Our set S of operations has the following two elements:

`add(x):`	add x to the array if array is not full
`resize():`	create new array twice the size and copy elems

NOTE: `resize` is called only when `add` has been called but array is full.

Actual cost function. The cost function is defined by

$$c(\texttt{add}) = 1$$
$$c(\texttt{resize}) = 3k \quad \text{if current array has k elements}$$

NOTE: Resizing requires $2k$ steps to create an array of twice the size, and k more steps to copy elements from old array to new array.

Amortized cost function. The amortized cost function we will use is defined by

$$\hat{c}(\texttt{add}) = 7$$

$$\hat{c}(\texttt{resize}) = 0$$

Consider the first few steps of a sequence of operations:

1. x_0 is added in position 0 at cost of \$1; at position 0 we have \$6.
2. `add` cannot be performed so `resize` is invoked; new array has size 2 and costs \$3; this leaves \$3 at pos. 0
3. `add` is performed; at position 1 we have \$6.
4. `add` cannot be performed so `resize` is invoked; this costs \$6, leaving \$0 in position 1
5. two more `add`s can be done, giving \$6 in positions 2 and 3
6. next `resize` costs \$12, leaving \$0 everywhere but initial \$3 in position 0

In general:

For a fixed natural number n, we prove the following statement $P(i)$ by induction on $i, 1 \leq i \leq n$: The ith `resize` operation produces an array of length 2^i which can be paid for by the dollars accumulated in the right half of the current array.

Clearly $P(1)$ is true. Assume $P(i)$ holds true, where $i < n$; we prove $P(i+1)$. We begin by describing the state of the array at the time of the $i+1$st `resize`. By the induction hypothesis, the ith resize produced an array of length 2^i. To arrive at the next `resize`, the `add` operation will be invoked to fill the right half of the new array. A total of 2^{i-1} new elements are added in this way, each producing \$6 profit, for a total of $6 \cdot 2^{i-1}$ dollars. `resize` operation $i+1$ then produces a new array of size 2^{i+1} and requires 2^i elements to be copied. These steps cost $3 \cdot 2^i = 6 \cdot 2^{i-1}$ dollars. Therefore, this `resize` operation is paid for by the dollars accumulated in the right half of the current array.

Conclusion. We have shown that our cost function bounds the set of operations of adding and resizing. The total amortized cost of n operations is $\leq 7n$ which is $\mathrm{O}(n)$. Therefore, the amortized running time for a resize operation is $\mathrm{O}(n)/n = \mathrm{O}(1)$.

Resizing Using Fixed Increments. We have seen the amortized running time for resizing an array using the size-doubling strategy is O(1). If fixed increments are used for resizing, it can be shown that the average running time for adding n elements to an initially empty array (of length 0) is $\Omega(n^2)$. Therefore, the average cost of a single resize operation in this case is $\Omega(n)$.

The idea of the proof is easier to understand in an example. Suppose the fixed increment is 10, and we wish to compute the running time for adding n elements to an initially empty array, performing resize operations whenever the array is full. We compute the number of steps required at each resize — running time will be at least as big as total number of such steps.

resize #0 — 10 steps
resize #1 — $2 \cdot 10 + 10$ steps
resize #2 — $2 \cdot 20 + 10$ steps
resize #3 — $2 \cdot 30 + 10$ steps
$\ldots$

These obserevations lead to the following bound for the fixed-increment algorithm with increment size = 10:

$$\text{running time} \;\geq\; \sum_{i=1}^{n/10} (20i + 10) = n^2 + n,$$

which is $\Omega(n^2)$.