Insertion Sort: Analysis of Complexity

Georgy Gimel'farb

COMPSCI 220 Algorithms and Data Structures

1. Worst-case complexity of insertion sort
2. Average-case, or expected complexity of insertion sort
3. Analysis of inversions
4. Selection and bubble sort of complexity $\Theta(n^2)$

Analysing Complexity of Insertion Sort

Iterative growth of a head ("sorted" sublist) of a list $\mathbf{A}$:

$$\underbrace{\boxed{a[0]}\boxed{a[1]}\boxed{\ldots}\boxed{a[i-1]}}_{\textbf{Head}\text{ (sorted sublist) of size } i}\ \underbrace{\boxed{a[i]}\boxed{a[i+1]}\boxed{\ldots}\boxed{a[n-1]}}_{\textbf{Tail}\text{ (unsorted sublist) of size } n-i}$$

$n-1$ iterations (stages) $i = 1, 2, \ldots, n-1$;
j; $1 \leq j \leq i$, comparisons and j or $j-1$ moves per stage:

1. **Initialisation:** the head sublist of size 1.
2. **Iteration:** until the tail sublist is empty, repeat:
   1. Choose the first element, $x = a[i]$ in the tail sublist.
   2. Find the last element, $y = a[j]$; $1 \leq j \leq i-1$, in the head sublist not exceeding x.
   3. Insert x after y in the head sublist.

Insertion sort is **correct**, since the head sublist is always sorted, and eventually expands to include all elements of $\mathbf{A}$.

Best- and Worst-case Complexity of Insertion Sort

The first element, $a[i]$, of the tail is moved to the correct position in the head by exhaustive backward search, comparing it to each element, $a[i-1], \ldots$, of the head until finding the right place.

The best case, $\Theta(n)$: if the inputs $\mathbf{A}$ are already in sorted order: $a[0] < a[1] < \ldots < a[n-1]$, i.e. $\mathbf{A} = \{1, 2, 3, 4\}$.

- One comparison and no moves per stage i; $i = 1, \ldots, n-1$.
- Comparisons in total: $1 + 1 + \ldots + 1 = n - 1 \in \Theta(n)$.

The worst case, $\Theta(n^2)$: if the inputs $\mathbf{A}$ contain distinct items in reverse order: $a[0] > a[1] > \ldots > a[n-1]$, i.e. $\mathbf{A} = \{4, 3, 2, 1\}$

- i comparisons and i moves per stage i; $i = 1, \ldots, n-1$.
- Comparisons in total: $1 + 2 + \ldots + n - 1 = \frac{(n-1)n}{2} = \frac{n^2 - n}{2} \in \Theta(n^2)$.

Average-case Complexity of Insertion Sort

Lemma 2.3, p.30

The average-case time complexity of insertion sort is $\Theta(n^2)$

The proof's outline:

- Assuming all possible inputs are equally likely, evaluate the average, or expected number $\overline{C}_i$ of comparisons at each stage $i = 1, \ldots, n-1$.
- Calculate the average total number $\overline{C} = \sum\limits_{i=1}^{n-1} \overline{C}_i$.
- Evaluate the average-case complexity of insertion sort by taking into account that the total number of data moves is at least zero and at most the number of comparisons.

Average Complexity of Insertion Sort at Stage i

$i+1$ positions in the already ordered head $a[0],\ldots,a[i-1]$ of a list $\mathbf{A}$ to insert the next unordered yet item $a[i]$:

$a[0]$	$a[1]$	$a[2]$	$\cdots$	$a[i-1]$	$a[i]$
$\uparrow_{j=0}$	$\uparrow_{1}$	$\uparrow_{2}$	$\uparrow_{\ldots}$	$\uparrow_{i-1}$	$\uparrow_{j=i}$
$C_{i:0}=i$	$C_{i:1}=i$	$C_{i:2}=i-1\cdots$		$C_{i:i-1}=2$	$C_{i:i}=1$
$M_{i:0}=i$	$M_{i:1}=i-1$	$M_{i:2}=i-2\cdots$		$M_{i:i-1}=1$	$M_{i:i}=0$

$C_{i:j}=i-j+1$ comparisons and $M_{i:j}=i-j$ moves to place $a[i]$ into each preceding position $j=i,i-1,\ldots,1$.

- $C_{i:i}=i$ comparisons and $M_{i:i}=i$ moves for $j=0$.

Average number, $\overline{C}_i=\frac{1}{i+1}\sum_{j=0}^{i}C_{i:j}$, of comparisons at stage i:

$$\overline{C}_i=\frac{1+2+\ldots+i+i}{i+1}=\frac{\frac{i(i+1)}{2}+i}{i+1}=\frac{i}{2}+\frac{i}{i+1}\equiv\frac{i}{2}+\left(1-\frac{1}{i+1}\right)$$

Total Average Complexity for n Input Items

The total average number of comparisons for $n-1$ stages:

$$\overline{C} = \overbrace{\left(\frac{1}{2}+\left(1-\frac{1}{2}\right)\right)}^{\overline{C}_1} + \overbrace{\left(\frac{2}{2}+\left(1-\frac{1}{3}\right)\right)}^{\overline{C}_2} + \ldots + \overbrace{\left(\frac{n-1}{2}+\left(1-\frac{1}{n}\right)\right)}^{\overline{C}_{n-1}}$$

$$= \frac{1}{2}\underbrace{(1+2+\ldots+(n-1))}_{\frac{(n-1)n}{2}} +$$

$$\underbrace{\left(1-\frac{1}{2}\right)+\left(1-\frac{1}{3}\right)+\ldots+\left(1-\frac{1}{n}\right)}_{(n-1)-(H_n-1)=n-H_n}$$

$$= \frac{(n-1)n}{4} + n - H_n \in \Theta(n^2)$$

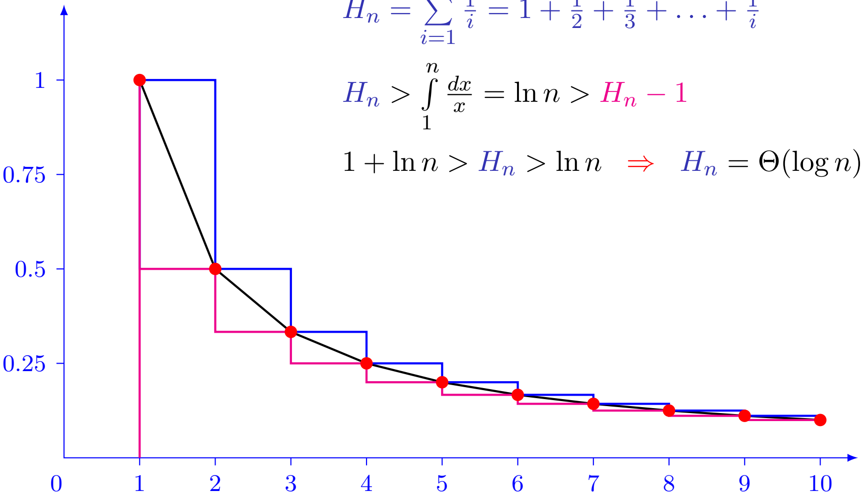
where $H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n$ when $n \to \infty$ is the n-th harmonic number.

Math Appendix: Evaluating Harmonic Numbers

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{i}$$

$$H_n > \int_1^n \frac{dx}{x} = \ln n > H_n - 1$$

$$1 + \ln n > H_n > \ln n \quad \Rightarrow \quad H_n = \Theta(\log n)$$



Analysis of Inversions

The running time of insertion sort is strongly related to **inversions** in a list $\mathbf{A}$ to be sorted.

Definition 2.5: An inversion in a list $\mathbf{A} = [a_1, a_2, \ldots, a_n]$ is any ordered pair of positions (i, j) such that $i < j$ but $a_i > a_j$.

Examples of inversions: $[\ldots, 2, \ldots, 1]$ or $[100, \ldots, 35, \ldots]$.

List $\mathbf{A}$	Number of inversions	Reverse list $\mathbf{A}_{\text{rev}}$	Number of inversions	Total
$[3, 2, 5]$	1	$[5, 2, 3]$	2	3
$[3, 2, 5, 1]$	4	$[1, 5, 2, 3]$	2	6
$[1, 2, 3, 5, 7]$	0	$[7, 5, 3, 2, 1]$	10	10

The number of inversions measures how far a list is from being sorted.

Analysis of Inversions

Number of inversions I_i, comparisons C_i and data moves M_i for each element $a[i]$ in $\mathbf{A}$:

Element i	0	1	2	3	4	5	6	
$\mathbf{A}$	44	13	35	18	15	10	20	
I_i		1	1	2	3	5	2	$I = 14$
C_i		1	2	3	4	5	3	$C = 18$
M_i		1	1	2	3	5	2	$M = 14$

Because $I_i = M_i$ is always true, the total number $I = \sum_{i=1}^{n-1} I_i$ of inversions is equal to the total number $M = \sum_{i=1}^{n-1} M_i$ of backward moves of elements $a[i]$ during the sort.

Analysis of Inversions

The total number of data comparisions $C = \sum_{i=1}^{n-1} C_i$ is also equal to the total number of inversions plus at most $n-1$.

Total number of inversions in both an arbitrary list $\mathbf{A}$ and its reverse $\mathbf{A}_{\text{rev}}$ is equal to the **total number of the ordered pairs** $(i < j)$ of integers $i, j \in \{1, \ldots, n-1\}$:

$$\binom{n-1}{2} = \frac{(n-1)n}{2}$$

- A sorted list has no inversions.
- A reverse sorted list of size n has $\frac{(n-1)n}{2}$ inversions.
- In the average, all lists of size n have $\frac{(n-1)n}{4}$ inversions.

Complexity of Insertion Sort by Analysing Inversions

Exactly **one inversion** is removed by swapping two neighbours being out of order: $a_{i-1} > a_i$.

- If an original list has I inversions, insertion sort has to swap I pairs of neighbours.
- A list with I inversions results in $\Theta(n + I)$ running time of `insertionSort` because of $\Theta(n)$ other operations in the algorithm.
  - In the very rare best case of a nearly sorted list for which I is $\Theta(n)$, insertion sort runs in linear time.
  - The worst-case time: $c\frac{n^2}{2}$, or $\Theta(n^2)$.
  - The average-case, or expected time: $c\frac{n^2}{4}$, or still $\Theta(n^2)$.

More efficient sorting algorithms must eliminate more than just one inversion between neighbours per swap.

Implementation of Insertion Sort

The number of comparisons does not depend on how the list is implemented, but the number of moves does.

- Backward moves in an array implementation of a list:
    - Shifting elements to the right (linear time per stage) in the worst and average case, or
    - Successive swaps to move the element backward.
- Insertion operation in a linked list implementation of a list:
    - Constant-time insertion of an element.
    - Fewer swaps by simply scanning backward (but it may take time for a singly linked list).

None of the implementation issues affect the asymptotic $\Theta(n^2)$ running time of the algorithm, just the hidden constants and lower order terms, due to too many comparisons in the worst and average cases.

Quadratic $\Theta(n^2)$ Selection Sort: Java Code

```
// Selection sort of an input array a of size n:
// building a head by successive minima selection in a tail
//
// Each leftmost unordered a[i] is swapped with the minimum element
// selected among the unordered yet elements a[i+1],...,a[n-1]

public static void selectionSort( int [ ] a ) {
  for ( int i = 1; i < a.length - 1; i++ ) {
      int posMin = i;
        // for-loop for selecting position of the minimum element
      for ( int k = i + 1; k < a.length; k++ ) {
        if ( a[ posMin ] > a[ k ] ) posMin = k;
      }
      if ( posMin != i ) swap( a, i, posMin );
       // swap a[i] with the minimum element selected
  }
}
```

Quadratic $\Theta(n^2)$ Bubble Sort: Java Code

```
// Bubble sort of an input array a of size n:
// n - 1 iterations to bubble up the maximum element
// among the unordered yet elements a[0],...,a[i]
//
// Each iteration i performs successive bottom-up swaps of
// the larger element in each adjacent pair of the elements
// for bubbling up the maximal element from a[0],...,a[i]

public static void bubbleSort( int [ ] a ) {
  for ( int i = a.length - 1; i > 0; i-- ) {
     for ( int k = 0; k < i; k++ ) {
       if ( a[ k ] > a[ k + 1 ] )
                                   swap( a, k, k + 1 );
              bubble up the larger of the two adjacent elements
     }
  }
}
```