Data Structures and Algorithms

Prof. Ganesh Ramakrishnan,
Prof. Ajit Diwan,
Prof. D.B. Phatak

Department of Computer Science and Engineering
IIT Bombay

Session: Sorted List-Based(Insertion Sort)

Sorted List-based (Insertion Sort)

- $insert_i(e, P)$: Scan list and insert in order to maintain increasing order of list P
  - $1 + 2 + \ldots n = O(n^2)$
- $delete(P, m)$: Remove the (min) element m at the beginning of the list P
  - $O(n)$
- $\Rightarrow O(n^2)$ overall

Array-based in-place version of Insertion Sort

Algorithm Insertion-Sort(S)
Input: Array S
Output: Array S sorted in increasing order
$i = 1$, $n = length(S)$
repeat
  1. $k = S[i]$
  //Insert k into the sorted array $S[1 \ldots i-1]$
  2. $j = i - 1$
  while $j > 0$ and $S[j] > k$ **do**
    1. $S[j+1] = S[j]$
    2. $j = j - 1$
  end while
  $S[j+1] = k$
until $i > n$

Insertion Sort & Loop Invariant

- Algorithm maintains following loop invariant:
  - At start of each iteration of i^{th} **repeat** loop, the subarray $S[1 \ldots i-1]$ consists of the elements of the old $S[1 \ldots i-1]$ but sorted in ascending order

Analysis of in-place Insertion Sort

Algorithm Insertion-Sort(S)
Input: Array S
Output: Array S sorted in increasing order
$i = 1, n = length(S)$ $\Longrightarrow$ c_1 $\times$ 1 times
repeat
  1. $k = S[i]$ $\Longrightarrow$ c_2 $\times$ $n - 1$ times
  //Insert k into the sorted array $S[1 \ldots i-1]$
  2. $j = i - 1$ $\Longrightarrow$ c_3 $\times$ $n - 1$ times
  while $j > 0$ and $S[j] > k$ **do**
    1. $S[j+1] = S[j]$ $\Longrightarrow$ c_4 $\times \sum_{i=2}^{n} (w_i - 1)$ times
    2. $j = j - 1$ $\Longrightarrow$ c_5 $\times \sum_{i=2}^{n} (w_i - 1)$ times
  end while $\Longrightarrow$ c_6 $\times \sum_{i=2}^{n} w_i$ times
  $S[j+1] = k$ $\Longrightarrow$ c_7 $\times$ $n - 1$ times
until $i > n$ $\Longrightarrow$ c_8 $\times$ n times

Figure: w_i is number of iterations of the inner while loop.

Analysis of in-place Insertion Sort

Algorithm Insertion-Sort(S)
Input: Array S
Output: Array S sorted in increasing order
$i = 1, n = length(S) \implies c_1 \quad \times\ 1$ times
repeat
1. $k = S[i] \implies c_2 \quad \times\ n-1$ times
//Insert k into the sorted array $S[1 \ldots i-1]$
2. $j = i - 1 \implies c_3 \quad \times\ n-1$ times

while $j > 0$ and $S[j] > k$ **do**

1. $S[j+1] = S[j] \implies c_4 \quad \times \sum_{i=2}^{n}(w_i - 1)$ times
2. $j = j - 1 \implies c_5 \quad \times \sum_{i=2}^{n}(w_i - 1)$ times

end while $\implies c_6 \quad \times \sum_{i=2}^{n} w_i$ times

$S[j+1] = k \implies c_7 \quad \times\ n-1$ times
until $i > n \implies c_8 \quad \times\ n$ times

Figure: w_i is number of iterations of the inner while loop.

$$T(n) = c_1 + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n}(w_i - 1) + c_5 \sum_{i=2}^{n}(w_i - 1) + c_6 \sum_{i=2}^{n} w_i + c_7(n-1) + c_8 n$$

Analysis of in-place Insertion Sort

- $T(n) = c_1 + c_2(n-1) + c_3(n-1) + c_4\sum_{i=2}^{n}(w_i - 1) + c_5\sum_{i=2}^{n}(w_i - 1) + c_6\sum_{i=2}^{n} w_i + c_7(n-1) + c_8 n$
- **Best Case Running Time:**

Analysis of in-place Insertion Sort

- $T(n) = c_1 + c_2(n-1) + c_3(n-1) + c_4\sum_{i=2}^{n}(w_i - 1) + c_5\sum_{i=2}^{n}(w_i - 1) + c_6\sum_{i=2}^{n} w_i + c_7(n-1) + c_8 n$
- **Best Case Running Time:**
  - If $S[1 \ldots n]$ is already sorted, $w_i = 1$ for $i = 2 \ldots n$
  - $\implies T(n) = c_1 + c_2(n-1) + c_3(n-1) + c_6(n-1) + c_7(n-1) + c_8 n = c_1 + n(c_2 + c_3 + c_6 + c_7 + c_8) - (c_2 + c_3 + c_6 + c_7) = O(n)$

Analysis of in-place Insertion Sort

- $T(n) = c_1 + c_2(n-1) + c_3(n-1) + c_4\sum_{i=2}^{n}(w_i - 1) + c_5\sum_{i=2}^{n}(w_i - 1) + c_6\sum_{i=2}^{n} w_i + c_7(n-1) + c_8 n$
- **Best Case Running Time:**
  - If $S[1\ldots n]$ is already sorted, $w_i = 1$ for $i = 2\ldots n$
  - $\implies T(n) = c_1 + c_2(n-1) + c_3(n-1) + c_6(n-1) + c_7(n-1) + c_8 n = c_1 + n(c_2 + c_3 + c_6 + c_7 + c_8) - (c_2 + c_3 + c_6 + c_7) = O(n)$
- **Worst Case Running Time:**

Analysis of in-place Insertion Sort

- $T(n) = c_1 + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n}(w_i - 1) + c_5 \sum_{i=2}^{n}(w_i - 1) + c_6 \sum_{i=2}^{n} w_i + c_7(n-1) + c_8 n$
- **Best Case Running Time:**
  - If $S[1 \ldots n]$ is already sorted, $w_i = 1$ for $i = 2 \ldots n$
  - $\implies T(n) = c_1 + c_2(n-1) + c_3(n-1) + c_6(n-1) + c_7(n-1) + c_8 n = c_1 + n(c_2 + c_3 + c_6 + c_7 + c_8) - (c_2 + c_3 + c_6 + c_7) = O(n)$
- **Worst Case Running Time:**
  - If $S[1 \ldots n]$ is sorted in reverse order, $w_i = i$ for $i = 2 \ldots n$
  - $\implies T(n) = c_1 + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n-1)}{2}\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n+1)}{2} - 1\right) + c_7(n-1) + c_8 n = O(n^2)$
- How about the average case?

Average Complexity of in-place Insertion Sort

- A **swap** in insertion sort: $S[j+1] = S[j]$ followed by $S[j+1] = k$
- A **swap** occurs for each **inversion**:
  - An ordered pair of indices (i, j) into S is an inversion if $i < j$ but $S[i] > S[j]$
- Number of operations $T(n) =$ time to scan each element $+$ number of inversions
- $I =$ number of inversions $\longrightarrow T(n) = O(n + I)$

Average Complexity of in-place Insertion Sort

- A **swap** in insertion sort: $S[j+1] = S[j]$ followed by $S[j+1] = k$
- A **swap** occurs for each **inversion**:
  - An ordered pair of indices (i, j) into S is an inversion if $i < j$ but $S[i] > S[j]$
- Number of operations $T(n)$ = time to scan each element + number of inversions
- I = number of inversions $\longrightarrow T(n) = O(n + I)$
- **Claim:** Average number of inversions in a list of n distinct elements is $\frac{n(n-1)}{4}$

Average number of Inversions

- Let S_r be the reverse of S.
- $\Rightarrow$ A pair of elements e_1 and e_2 will be inverted in exactly one of S and S_r
- $\Rightarrow$ Total number of such pairs inverted in one of S or $S_r = \frac{n(n-1)}{2}$
- $\Rightarrow$ Total number of inversions across all permutations $= \frac{n!}{2} \times \frac{n(n-1)}{2}$

Average number of Inversions

- Let S_r be the reverse of S.
- $\Rightarrow$ A pair of elements e_1 and e_2 will be inverted in exactly one of S and S_r
- $\Rightarrow$ Total number of such pairs inverted in one of S or $S_r = \frac{n(n-1)}{2}$
- $\Rightarrow$ Total number of inversions across all permutations $= \frac{n!}{2} \times \frac{n(n-1)}{2}$
- $\Rightarrow$ Average number of inversions I across all permutations $= \frac{Total}{n!} = \frac{\frac{n!}{2} \times \frac{n(n-1)}{2}}{n!} = \frac{n(n-1)}{4}$

Average number of Inversions

- Let S_r be the reverse of S.
- $\Rightarrow$ A pair of elements e_1 and e_2 will be inverted in exactly one of S and S_r
- $\Rightarrow$ Total number of such pairs inverted in one of S or $S_r = \frac{n(n-1)}{2}$
- $\Rightarrow$ Total number of inversions across all permutations $= \frac{n!}{2} \times \frac{n(n-1)}{2}$
- $\Rightarrow$ Average number of inversions I across all permutations $= \frac{Total}{n!} = \frac{\frac{n!}{2} \times \frac{n(n-1)}{2}}{n!} = \frac{n(n-1)}{4}$
- $\Rightarrow$ Average $T(n) = \frac{n(n-1)}{4} + n = O(n^2)$
  - True about **ANY** swap based algorithm, since swap removes exactly one inversion in a single step.

Summary Analysis of in-place Insertion Sort

- **Best Case Running Time:** $S[1 \ldots n]$ is already sorted $\implies T(n) = O(n)$
- **Worst Case Running Time:** $S[1 \ldots n]$ is sorted in reverse order $\implies T(n) = O(n^2)$
- **Average Case Running Time:** $T(n) = O(n^2)$
- Can an alternative algorithm give better average and worst-case running times?

- Next Session: Heap Based $\Rightarrow$ Heap Sort

Thank you