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)

Insertion

S=[4 5 1 6 2]   P=[ ]   P=[4]   P=[4 5]   P=[1 4 5] . . .

- $insert_i(e, P)$: 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

Deletion

P=[1 2 4 5 6]   S=[1 2 4 5 6]

Array-based in-place version of Insertion Sort

P will form a prefix of S

S= [4 5 1 2 3]

Algorithm Insertion-Sort(S)
Input: Array S
Output: Array S sorted in increasing order
$i = 1$, $n = length(S)$
repeat
  1. $k = S[i]$ k=5 (when i=2)
  //Insert k into the sorted array $S[1 \ldots i-1]$ = P = [4] (when i=2)
  2. $j = i - 1$ j=1 (scan P starting from right most element)
  while $j > 0$ and $S[j] > k$ **do**
    1. $S[j+1] = S[j]$ → shift the element of P to the right to make space for k until you find S[j] ≤ k
    2. $j = j - 1$
  end while
  $S[j+1] = k$
until $i > n$

eg: S= [4 5 1 2 3] ⇒ S= [1 4 5 2 3]

Insertion Sort & Loop Invariant

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

mimics priority queue P

1. Initialization:
2. Maintainance : Holds since $S[1..j]$ is sorted & $S[j] \leq K$ & since $S[j+1] = K$ & $S[j+2..i]$ are sorted & shifted
3. Termination

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

Variable # of right shifts until $S[j] \leq k$

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
$i = 1, n = length(S)$ $\Longrightarrow$ 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.

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

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:**

$S = [1 \quad 2 \quad 3 \quad 4]$

Analysis of in-place Insertion Sort

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

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:**

$S = [5\ 4\ 2\ 3\ 1]$ $\quad w_2 = 2 \quad w_3 = 3 \; \ldots$

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

Average Complexity of in-place Insertion Sort

realized through a series of shifts & a substitution

1 2 3 4 3 2

- 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 $\longrightarrow T(n) = O(n + I)$

For worst case $I = O(n^2)$

For best case $I = 0$

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

Factoring out redundancies in counting by clubbing together S & its reverse

- 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 $S_r = \frac{n(n-1)}{2}$ $= {}^nC_2$
- $\Rightarrow$ Total number of inversions across all permutations $= \frac{n!}{2} \times \frac{n(n-1)}{2}$

\# of permutations × ${}^nC_2$

factoring out the reverse of each permutation

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 $= \underbrace{\frac{n!}{2} \times \frac{n(n-1)}{2}}_{\text{\# total inv}}$
- $\Rightarrow$ Average number of inversions I across all permutations $= \frac{Total}{n!} = \frac{\frac{n!}{2} \times \frac{n(n-1)}{2}}{n!} = \frac{n(n-1)}{4}$

# of perm.

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:** $S[1 \dots n]$ is already sorted $\implies T(n) = O(n)$
- **Worst Case Running Time:** $S[1 \dots n]$ 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?

Can we avoid so many swaps

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

Thank you