Getting Started

Contents

- Sorting problem
- 2 sorting algorithms
  - Insertion sort
  - Merge sort

Sorting problem

keys

- **Input**
  - A sequence of n numbers $<a_1, a_2, \ldots, a_n>$.
- **Output**
  - A permutation (reordering) $<a'_1, a'_2, \ldots, a'_n>$ of the input sequence such that $a'_1 \le a'_2 \le \ldots \le a'_n$.
- e.g.>
  - Input: < 5, 2, 4, 6, 1, 3>
  - Output: < 1, 2, 3, 4, 5, 6>

Insertion sort

- **Insertion sort**
  - Description
  - Correctness
  - Performance

Description

- What is insertion sort?
  - A sorting algorithm using insertion.

- What is insertion?
  - Given a key and a sorted list of keys, insert the key into the sorted list preserving the sorted order.
  - e. g.> Insert 3 into <1, 2, 4, 5, 6>

Description

- Insertion sort uses insertion incrementally.
  - Let $A[1..n]$ denote the array storing keys.
  - Insert $A[2]$ into $A[1]$.
  - Insert $A[3]$ into $A[1..2]$.
  - Insert $A[4]$ into $A[1..3]$.

    .

    .

    .
  - Insert $A[n]$ into $A[1..n-1]$.

Description: example

- 5 2 4 6 1 3
- 5 2 4 6 1 3
- 2 5 4 6 1 3
- 2 4 5 6 1 3
- 2 4 5 6 1 3
- 1 2 4 5 6 3
- 1 2 3 4 5 6

Description: pseudo code

Pseudocode conventions are given in p. 19 - 20 of the textbook.

INSERTION-SORT(A)

for $j = 2$ **to** $A.length$

$key = A[j]$

$i = j - 1$

while $i > 0$ and $A[i] > key$

$A[i + 1] = A[i]$

$i = i - 1$

$A[i + 1] = key$

n-1 iterations of insertion.

Insert $A[j]$ into $A[1..j - 1]$.

Find a place to put $A[j]$.

Put $A[j]$.

Insertion sort

- Insertion sort
  - Description
  - Correctness
  - Performance
    - Running time
    - Space consumption

Running time

- How to analyze the running time of an algorithm?
  - Consider running the algorithm on a specific machine and measure the running time.
    - We cannot compare the running time of an algorithm on a machine with the running time of another algorithm on another machine.
    - So, we have to measure the running time of every algorithm on a specific machine, which is impossible.
  - Hence, we count the number of instructions used by the algorithm.

Instructions

- Arithmetic
  - Add, Subtract, Multiply, Divide, remainder, floor, ceiling
- Data movement
  - Load, store, copy
- Control
  - Conditional branch
  - Unconditional branch
  - Subroutine call and return

Running time

- The running time of an algorithm grows with the input size, which is the number of items in the input.

- For example, sorting 10 keys is faster than sorting 100 keys.

- So the running time of an algorithm is described as a function of input size n, for example, $T(n)$.

Running time of insertion sort

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n - 1$
$i = j - 1$	c_4	$n - 1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i + 1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i + 1] = key$	c_8	$n - 1$

- ***T*(*n*)**: The sum of product of *cost* and *times* of each line.

Running time of insertion sort

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

cost	*times*
c_1	n
c_2	$n - 1$
c_4	$n - 1$
c_5	$\sum_{j=2}^{n} t_j$
c_6	$\sum_{j=2}^{n} (t_j - 1)$
c_7	$\sum_{j=2}^{n} (t_j - 1)$
c_8	$n - 1$

- ***T*(*n*)**: The sum of product of *cost* and *times* of each line.

Running time of insertion sort

- t_j: The number of times the **while** loop test is executed for j.

- Note that **for**, **while** loop test is executed one time more than the loop body.

Running time of insertion sort

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

- Although the size of the input is the same, we have
  - best case
  - average case, and
  - worst case.

Running time of insertion sort

- Best case
  - If $A[1..n]$ is already sorted, $t_j = 1$ for $j = 2, 3, \ldots, n$.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + \cancel{c_6 \sum_{j=2}^{n} (t_j - 1)}$$

$$+ \cancel{c_7 \sum_{j=2}^{n} (t_j - 1)} + c_8(n-1)$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

- This running time can be expressed as $an+b$ for *constants a* and b; it is thus a ***linear function*** of n.

Running time of insertion sort

- Worst case
  - If $A[1..n]$ is sorted in reverse order, $t_j = j$ for $j = 2, 3, \ldots, n$.

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \quad \text{and} \quad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right)$$

$$+ c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

- This running time can be expressed as $an^2 + bn + c$ for constants a, b, and c; it is thus a ***quadratic function*** of n.

Running time of insertion sort

- **Only the degree of leading term is important.**
  - Because we are only interested in the ***rate of growth*** or ***order of growth.***
  - For example, a quadratic function grows faster than any linear function.

- The degree of leading term is expressed as Θ–notation.
  - The worst-case running time of insertion sort is $\Theta(n^2)$.

Space consumption of insertion sort

- $\Theta(n)$ space.
- Moreover, the input numbers are ***sorted in place*.**
  - $n + c$ space for some constant c.
- A sorting algorithm is ***stable*** if two items compare as equal, then their relative order(input) will be preserved(output).

Self-study on Insertion Sort

- **Exercise 2.1-1**
- **Exercise 2.1-2**