10. Insertion Sort

Insertion Sort

It is said that insertion sort is how many sort a hand of cards. An empty hand is sorted. A single, first card, is sorted. As every other card is dealt, insert it into the previously sorted hand.

The functional `insertion_sort` code below uses a helper function to perform the insertion.

To insertion sort a list `x:xs`,

- Insertion sort the tail `xs`.
- Insert `x` at the head if it is smaller than the head of the insertion sorted list `(y:ys)`.
- Otherwise, leave `y` as the head of the sorted list and insert `x` in the tail `ys`.

Listing 28: Functional Insertion Sort

115a ⟨*Functional insertion sort* 115a⟩≡

```
insertion_sort         :: Ord a => [a] -> [a]
insertion_sort []      = []
insertion_sort [x]     = [x]
insertion_sort (x:xs) = insert (insertion_sort xs)
    where insert [] = [x]
          insert (y:ys) | x <= y    = x : y : ys
                        | otherwise = y : insert ys
```

An imperative implementation is given below. It assumes a *sentinel*, $A[0]$, that is smaller every other element in the array. The invariant of the algorithm is that after the ith step, the array $A[0..i]$ is sorted.

Listing 29: Imperative Insertion sort

115b ⟨*Insertion sort* 115b⟩≡

```
void InsertionSort(int A[], int n) {
  int i, j, v;
  for (i = 2; i < n; i++) {
    v = A[i];
    j = i;
    while (A[j-1] > v) {
      A[j] = A[j-1];
      j = j - 1;
    }
    A[j] = v;
```

```
    }
  }
```

Insertion Sort – Analysis of Complexity

The time complexity of the imperative algorithm can be computed using these observations:

- The outer `for` loop on i is executed $n-1$ times
- There are two initial assignment in this outer loop, a `while` loop, and a final assignment
- The comparison in the `while` loop Boolean expression may execute as many as i times and as few as 0 times
- In the *best case* (the data is sorted in ascending order), insertion sort will execute $3n-3$ assignments and $n-1$ tests of the Boolean expression in the `while` loop. Thus, the time complexity is $O(n)$
- In the *worst case* (the data is in descending order), the `while` loop executes i times making i evaluations of the Boolean expression and 2 assignments on each pass of the loop. This occurs for every value of i from 2 to n, thus the total complexity is

  $$3n-3+\sum_{i=2}^{n} 3i = 3n-3+3\left[\frac{n(n+1)}{2}-1\right]$$

  The time complexity is $O(n^2)$
- For the *average case*, we need the probability that k compares are made in the `while` test for $k=1$ to i
- For given $k=1,\ldots,i$ there will be k compares if and only if

  $$A[i-1] > A[i],\ A[i-2] > A[i], \ldots, A[i-k+1] > A[i]$$

  and

  $$A[i-k] \leq A[i]$$
- That is, $A[i]$ is the kth largest element in the array

  $$A[1],\ A[2],\ \ldots,\ A[i-1],\ A[i]$$
- The probability of this is $1/i$: There is one out of i positions to place the kth largest element

- Thus the average number of comparisons is

$$\begin{aligned}\sum_{i=2}^{n}\sum_{k=1}^{i}\frac{1}{i}(k) &= \sum_{i=2}^{n}\frac{i+1}{2}\\ &= \frac{(n+1)(n+2)}{4}-\frac{3}{2}\end{aligned}$$

- The average case complexity is $O(n^2)$

Example: Insertion sort operations

$i=2$	*order*	*compares*	*reordered*	$i=3$	*compares*
	0 2 5 7	*1*			*1*
	0 2 7 5	*1*			*2*
	0 5 2 7	*2*	*0 2 5 7*		*1*
	0 5 7 2	*1*			*3*
	0 7 2 5	*2*	*0 2 7 5*		*2*
	0 7 5 2	*2*	*0 5 7 2*		*3*

For $i=2$, there is $\frac{9}{6}$ comparisons, on average
For $i=3$, there are $\frac{12}{6}$ comparisons, on average
The average number of comparisons, over all i, is

$$21/6 = 7/2 = \frac{(3+1)(3+2)}{4}-\frac{3}{2} = 5-\frac{3}{2}$$

Exercises

1. Consider the selection sort algorithm below. Explain the code and analyze its time complexity.

Listing 30: Functional Selection Sort

117 ⟨*Functional Selection sort* 117⟩≡

```
select_sort :: (Ord a) => [a] -> [a]
select_sort [] = []
select_sort xs = let x = maximum xs
  in select_sort (remove x xs) ++ [x]
  where remove _ [] = []
        remove a (x:xs)
          | x == a    = xs
          | otherwise = x : remove a xs
```

2. Consider the selection sort algorithm below.

Listing 31: Imperative selection sort

118 ⟨*Imperative selection sort* 118⟩≡

```
void selectionsort(int  A[], int n)
{
  int i, j, min;
  for (i = 1; i < n; i++) {
    min = i;
    for (j = i+1; j < n; j++) {
      if (A[j] < A[min]) {
        min = j;
        swap(A[min], A[i]);
      }
    }
  }
}
```

(a) Compute the average number of swaps on the 6 permutations of $\{2, 5, 7\}$. Construct a table, similar to the table in example to show your computations.

(b) Find a general formula for the average number of comparisons in the insertion sort algorithm.

Bibliography

Agrawal, M., Kayal, N., and Saxena, N. (2002). Primes is in P. http://www.cse.iitk.ac.in/news/primality.html. [page 205]

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340. [page 87]

Apostolico, A. and Galil, Z. (1997). *Pattern Matching Algorithms*. Oxford University Press. [page 53]

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press. [page 141]

Bentley, J. (1984a). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. [page 33]

Bentley, J. (1984b). Programming Pearls: How to sort. *Commun. ACM*, 27(4):287–ff.

Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772. [page 71]

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM. [page 199]

Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 135], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of Intractability*. W. H. Freeman. [page 193], [page 203]

Gödel, K. (1930). Die vollst andigkeit der axiome des logischen funktionenkalk uls. *Monatshefte f ur Mathematik und Physik*, 37:349–360. [page 194]

Gödel, K. (1992). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover books on advanced mathematics. Dover Publications. [page 194]

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press. [page 53]

Hoare, C. A. R. (1961). Quicksort. *Computer Journal*, 5(1):10–15.

Hofstadter, D. R. (1999). *Gödel, Escher, Bach: An Eternal Golden Braid. 20th Anniversary Edition*. Basic Books, New York, New York. [page 25]

Kernighan, B. W. and Ritchie, D. M. (1988). *The* C *Programming Language*. Prentice Hall, second edition. [page 31]

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111. [page 30]

Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, ACM Press. [page 147], [page 163]

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):240–267. [page 55]

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition. [page 31]

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley. [page 193], [page 195], [page 206]

Patterson, D. A. and Hennessy, J. L. (1996). *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann Publishers, Inc., second edition. [page 30]

Polya, G. (1945). *How To Solve It*. Princeton University Press. ISBN 0-692-02356-5. [page 26]

Rabhi, F. A. and Lapalme, G. (1999). *Algorithms: A Functional Programming Approach*. Addison-Wesley. [page 31]

Ramsey, N. (1994). Literate programming simplified. *IEEE Software*, 11(5):97 – 105. [page 30]

Sankoff, D. and Kruskal, J. B., editors (1983). *Time Warps, String Edits, and Macromolecules*. Addison-Wesley, Reading, Massachusetts. [page 146]

Sedgewick, R. (1977). The analysis of quicksort programs. *Acta Informatica*, 7:327–355.

Sedgewick, R. (1978). Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857.

Sedgewick, R. (2004). *Algorithms in Java*. Addison-Wesley. [page 31]

Shallit, J. (1994). Origins of the Analysis of the Euclidean Algorithm. *Historia Mathematica*, 21:401–419. [page 44]

Stansifer, R. (1984). Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department. [page 194]

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265. [page 197]

Wilf, H. (2006). *Generatingfunctionology: Third Edition*. AK Peters Series. Taylor & Francis. [page 19]

Index