In-place algorithm

From Wikipedia, the free encyclopedia

In computer science, an **in-place algorithm** (or in Latin **in situ**) is an algorithm which transforms input using a data structure with a small, constant amount of extra storage space. The input is usually overwritten by the output as the algorithm executes. An algorithm which is not in-place is sometimes called **not-in-place** or **out-of-place** (or **ex situ** in Latin).

An algorithm is sometimes, informally, called in-place as long as it overwrites its input with its output. In reality, this is not sufficient (as the case of quicksort demonstrates), nor is it necessary; the output space may be constant, or may not even be counted, for example if the output is to a stream. On the other hand, sometimes it may be more practical to count the output space in determining whether an algorithm is in-place, such as in the first reverse example below; this makes it difficult to strictly define in-place algorithms. In theory applications such as log-space reductions, it's more typical to always ignore output space (in these cases it's more essential that the output is *write-only*).

Contents

Examples

Given an array `a` of *n* items, suppose we want an array that holds the same elements in reversed order and dispose of the original. One seemingly simple way to do this is to create a new array of equal size, fill it with copies from `a` in appropriate order and then delete `a`.

```
function reverse(a[0..n - 1])
    allocate b[0..n - 1]
    for i from 0 to n - 1
        b[n − 1 − i] := a[i]
    return b
```

Unfortunately, this requires O(*n*) extra space for having the arrays `a` and `b` available simultaneously. Also, allocation and deallocation are often slow operations. Since we no longer need `a`, we can instead overwrite it with its own reversal using this in-place algorithm which will only need constant additional storage for the auxiliary variables `i` and `tmp`, no matter how large the array is.

```
function reverse_in_place(a[0..n-1])
    for i from 0 to floor((n-2)/2)
        tmp := a[i]
        a[i] := a[n − 1 − i]
        a[n − 1 − i] := tmp
```

As another example, there are a number of sorting algorithms that can rearrange arrays into sorted order in-place, including: Bubble sort, Comb sort, Selection sort, Insertion sort, Heapsort, Shell sort.

Quicksort operates in-place on the data to be sorted as it only ever swaps two elements. However, most implementations require O(log *n*) space to keep track of the recursive function calls as part of the divide and conquer strategy; so Quicksort is not an in-place algorithm.

Most selection algorithms are also in-place, although some considerably rearrange the input array in the process of finding the final, constant-sized result.

Some text manipulation algorithms such as trim and reverse may be done in-place.

In computational complexity

In computational complexity theory, in-place algorithms include all algorithms with O(1) space complexity, the class **DSPACE**(1). This class is very limited; it equals the regular languages.[1] In fact, it does not even include any of the examples listed above.

For this reason, we also consider algorithms in L, the class of problems requiring O(log *n*) additional space, to be in-place. Although this seems to contradict our earlier definition, we have to consider that in the abstract world our input can be arbitrarily large. On a real computer, a pointer requires only a small fixed amount of space, because the amount of physical memory is limited, but in general O(log *n*) bits are required to specify an index into a list of size *n*.

Does this mean quicksort is in-place after all? Not at all—technically, it requires O($\log^2 n$) space, since each of its O(log *n*) stack frames contains a constant number of pointers (each of size O(log *n*)).

Identifying the in-place algorithms with L has some interesting implications; for example, it means that there is a (rather complex) in-place algorithm to determine whether a path exists between two nodes in an undirected graph,[2] a problem that requires O(*n*) extra space using typical algorithms such as depth-first search (a visited bit for each node). This in turn yields in-place algorithms for problems such as determining if a graph is bipartite or testing whether two graphs have the same number of connected components. See SL for more information.

Role of randomness

In many cases, the space requirements for an algorithm can be drastically cut by using a randomized algorithm. For example, say we wish to know if two vertices in a graph of *n* vertices are in the same connected component of the graph. There is no known simple, deterministic, in-place algorithm to determine this, but if we simply start at one vertex and perform a random walk of about $20n^3$ steps, the chance that we will stumble across the other vertex provided that it's in the same component is very high. Similarly, there are simple randomized in-place algorithms for primality testing such as the Miller-Rabin primality test, and there are also simple in-place randomized factoring algorithms such as Pollard's rho algorithm. See RL and BPL for more discussion of this phenomenon.

In functional programming

Functional programming languages often discourage or don't support explicit in-place algorithms that overwrite data, since this is a type of side effect; instead, they only allow new data to be constructed. However, good functional language compilers will often recognize when an object very similar to an existing one is created and then the old one thrown away, and will optimize this into a simple mutation "under-the-hood".

Note that it is possible in principle to carefully construct in-place algorithms that don't modify data (unless the data is no longer being used), but this is rarely done in practice. See purely functional data structures.

See also

- Table of in-place and not-in-place sorting algorithms

References

1. ^ Maciej Liśkiewicz and Rüdiger Reischuk. The Complexity World below Logarithmic Space (http://citeseer.ist.psu.edu/34203.html). *Structure in Complexity Theory Conference*, pp. 64-78. 1994. Online: p. 3, Theorem 2.
2. ^ Omer Reingold. Undirected ST-connectivity in Log-Space (http://www.wisdom.weizmann.ac.il/~reingold/publications/sl.ps). Electronic Colloquium on Computational Complexity. No. 94.

Retrieved from "http://en.wikipedia.org/w/index.php?title=In-place_algorithm&oldid=621264319"

Categories: Algorithms