Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Esperanto
فارسی
Íslenska
Српски / srpski
Türkçe
Edit links

Patience sorting

From Wikipedia, the free encyclopedia

Patience sorting is a sorting algorithm, based on a solitaire card game, that has the property of being able to efficiently compute the length of a longest increasing subsequence in a given array.

Patience sorting

Class	Sorting algorithm
Data structure	Array
Worst case performance	$O(n \log n)$

Contents [hide]

The card game [edit]

The game begins with a shuffled deck of cards, labeled $1, 2, \ldots, n$.

The cards are dealt one by one into a sequence of piles on the table, according to the following rules.

1. Initially, there are no piles. The first card dealt forms a new pile consisting of the single card.
2. Each new card may be placed either on an existing pile whose top card has a value higher than the new card's value, thus increasing the number of cards in that pile, or to the right of all of the existing piles, thus forming a new pile.
3. When there are no more cards remaining to deal, the game ends.

The object of the game is to finish with as few piles as possible. D. Aldous and P. Diaconis[1] suggest defining 9 or fewer piles as a winning outcome for $n = 52$, which has approximately 5% chance to happen.

Algorithm for sorting [edit]

Given an n-element array with an ordering relation as an input for the sorting, consider it as a collection of cards, with the (unknown in the beginning) statistical ordering of each element serving as its index. Note that the game never uses the actual value of the card, except for comparison between two cards, and the relative ordering of any two array elements is known.

Now simulate the patience sorting game, played with the *greedy strategy*, i.e., placing each new card on the leftmost pile that is legally possible to use. At each stage of the game, under this strategy, the labels on the top cards of the piles are increasing from left to right. To recover the sorted sequence, repeatedly remove the minimum visible card.

Complexity [edit]

If values of cards are in the range $1, \ldots, n$, there is an efficient implementation with $O(n \cdot \log \log n)$ worst-case running time for putting the cards into piles, relying on a van Emde Boas tree. A description is given in the work by S. Bespamyatnikh and M. Segal.[2] This is slower than the $O(n)$ worst-case time for a simple pigeonhole sort for this range.

When no assumption is made about values, the greedy strategy can be implemented in $O(n \log n)$ comparisons in worst case. In fact, one can implement it with an array of stacks ordered by values of top cards and, for inserting a new card, use a binary search, which is $O(\log p)$ comparisons in worst case, where p is

the number of piles. To complete the sorting in an efficient way (aka $O(n \log n)$ worst case), each step will retrieve the card with the least value from the top of leftmost pile, and then some work has to be done. Finding the next card by searching it among all tops of piles, as in the wikibooks implementation suggested below, gives a $O(n\sqrt{n})$ worst case. However, we can use an efficient priority queue (for example, a binary heap) to maintain the piles so that we can extract the maximum data in $O(\log n)$ time. So, when patience sorting uses an efficient priority queue to perform one part of the algorithm, to achieve a worst-case time of $O(n \log n)$, that is the same worst-case time as when the entire sort is done using that queue, as in heapsort.

Algorithm for finding a longest increasing subsequence [edit]

First, execute the sorting algorithm as described above. The number of piles is the length of a longest subsequence. Whenever a card is placed on top of a pile, put a back-pointer to the top card in the previous pile (that, by assumption, has a lower value than the new card has). In the end, follow the back-pointers from the top card in the last pile to recover a decreasing subsequence of the longest length; its reverse is an answer to the longest increasing subsequence algorithm.

S. Bespamyatnikh and M. Segal[2] give a description of an efficient implementation of the algorithm, incurring no additional asymptotic cost over the sorting one (as the back-pointers storage, creation and traversal require linear time and space). They further show how to report *all* the longest increasing subsequences from the same resulting data structures.

Software Implementations [edit]

Python [edit]

This is an implementation of sorting an array using Patience Sort. The top cards are maintained as an ordered list. This is used to find the position of the pile where the next number should be placed. Piles are maintained as a list of list. The original array is iterated and each element is placed on the correct pile. Once the piles are created, the top number of each pile is inserted into a heap. Iteratively, the minimum element from the heap is removed and a new element from the corresponding pile is inserted into the heap.

The time complexity is $O(n \log n)$ and the space complexity is $O(n)$.

```
import bisect
import heapq

def find_pile(top_cards, n):
    """
    return the pile_id on which the
    number 'n' should be placed
    If no such pile exist return -1

    It also updates the list of top cards
    """
    pos = bisect.bisect_right(top_cards, n)
    if pos == len(top_cards):
        top_cards.append(n)
        return -1
    else:
        top_cards[pos] = n
        return pos

def patience_sort(a):
    top_cards = [] #maintain the list of top cards of each pile
    piles = [] #each pile will be a python list.

    for i in a:
        pile_id = find_pile(top_cards, i)
        if pile_id == -1:
            pile = [i] #create a new pile
            piles.append(pile)
        else:
            piles[pile_id].append(i)

    #piles are created now.
    #put the top cards of every pile in a heap
    heap = [(pile.pop(),pile_id) for pile_id,pile in enumerate(piles)]
```

```
    sorted_a = []
    while heap:
       i,pile_id  = heapq.heappop(heap)
       sorted_a.append(i)

       #get the next top_card from that pile:
       pile = piles[pile_id]
       if len(pile) > 0:
           i = pile.pop()
           heapq.heappush(heap, (i,pile_id))

    return sorted_a

def run():
    a = [2,6,3,1,5,9,2]
    sorted_a = patience_sort(a)
    print sorted_a

if __name__ == "__main__":
    run()
```

C++ [edit]

This is an implementation using Patience Sorting to sort an array, performing $O(n \log n)$ time complexity.

```
#include <vector>
#include <algorithm>
#include <stack>
#include <iterator>

template<typename PileType>
bool pile_less(const PileType& x, const PileType& y)
{
    return x.top() < y.top();
}

// reverse less predicate to turn max-heap into min-heap
template<typename PileType>
bool pile_more(const PileType& x, const PileType& y)
{
    return pile_less(y, x);
}

template<typename Iterator>
void patience_sort(Iterator begin, Iterator end)
{
    typedef typename std::iterator_traits<Iterator>::value_type DataType;
    typedef std::stack<DataType> PileType;
    std::vector<PileType> piles;

    for (Iterator it = begin; it != end; it++)
    {
        PileType new_pile;
        new_pile.push(*it);
        typename std::vector<PileType>::iterator insert_it =
            std::lower_bound(piles.begin(), piles.end(), new_pile,
                             pile_less<PileType>);
        if (insert_it == piles.end())
            piles.push_back(new_pile);
        else
            insert_it->push(*it);
    }
    // sorted array already satisfies heap property for min-heap

    for (Iterator it = begin; it != end; it++)
    {
        std::pop_heap(piles.begin(), piles.end(), pile_more<PileType>);
        *it = piles.back().top();
        piles.back().pop();
        if (piles.back().empty())
```

```
            piles.pop_back();
        else
            std::push_heap(piles.begin(), piles.end(), pile_more<PileType>);
    }
}
```

Clojure [edit]

Implementation using the Patience Sort approach. The elements (newelem) put on a pile combine the "card" with a reference to the top of the previous stack, as per the algorithm. The combination is done using cons, so what gets put on a pile is a list -- a descending subsequence. [3]

```
(defn place [piles card]
  (let [[les gts] (->> piles (split-with #(<= (ffirst %) card)))
        newelem (cons card (->> les last first))
        modpile (cons newelem (first gts))]
    (concat les (cons modpile (rest gts)))))

(defn a-longest [cards]
  (let [piles (reduce place '() cards)]
    (->> piles last first reverse)))

(println (a-longest [3 2 6 4 5 1]))
(println (a-longest [0 8 4 12 2 10 6 14 1 9 5 13 3 11 7 15]))
```

Output:

```
(2 4 5)
(0 2 6 9 11 15)
```

Java [edit]

```
import java.util.*;
public class PatienceSort
{
    public static <E extends Comparable<? super E>> void sort (E[] n)
    {
        List<Pile<E>> piles = new ArrayList<Pile<E>>();
        // sort into piles
        for (E x : n)
        {
            Pile<E> newPile = new Pile<E>();
            newPile.push(x);
            int i = Collections.binarySearch(piles, newPile);
            if (i < 0) i = ~i;
            if (i != piles.size())
                piles.get(i).push(x);
            else
                piles.add(newPile);
        }
        System.out.println("longest increasing subsequence has length = " +
piles.size());

        // priority queue allows us to retrieve least pile efficiently
        PriorityQueue<Pile<E>> heap = new PriorityQueue<Pile<E>>(piles);
        for (int c = 0; c < n.length; c++)
        {
            Pile<E> smallPile = heap.poll();
            n[c] = smallPile.pop();
            if (!smallPile.isEmpty())
                heap.offer(smallPile);
        }
        assert(heap.isEmpty());
    }
```

```
    private static class Pile<E extends Comparable<? super E>> extends Stack<E>
implements Comparable<Pile<E>>
    {
        public int compareTo(Pile<E> y) { return peek().compareTo(y.peek()); }
    }
}
```

Go [edit]

```
package main

import (
    "fmt"
    "container/heap"
)

type PileHeap [][]int

func (h PileHeap) Len() int      { return len(h) }
func (h PileHeap) Swap(i, j int) { h[i], h[j] = h[j], h[i] }
func (h PileHeap) Less(i, j int) bool {
    return h[i][len(h[i])-1] < h[j][len(h[j])-1]
}

func (h *PileHeap) Push(x interface{}) {
    *h = append(*h, x.([]int))
}

func (h *PileHeap) Pop() interface{} {
    old := *h
    n := len(old)
    x := old[n-1]
    *h = old[0 : n-1]
    return x
}

/*
bisectPilesRight uses binary search to returns the index where to insert card x,
assuming piles is already sorted according to the value of the top card
in each pile

The return value i is such that it's the largest i
for which the top card in piles[i] >= x and
return i == len(piles) if no such pile can be found
*/
func bisectPilesRight(piles [][]int , x int) int {
    lo, hi := 0, len(piles)
    for lo < hi {
        // invariant: x maybe between a[lo]...a[hi-1]
        mid := lo + (hi-lo)/2  // don't use (lo+hi)/2 to avoid overflow
        // Note that since (hi-lo)/2 >= 0, lo <= mid < hi
        pile := piles[mid]
        if x < pile[len(pile) - 1] { // compare x to top of pile
            hi = mid   // x may be between a[lo]...a[mid-1]
        } else {
            lo = mid+1 // x may be between a[mid+1]...a[hi]
        }
        // The new range is either lo...mid or mid+1...hi and
        // because lo<=mid<hi, the new range is always smaller than lo..hi
    }
    return lo
}


func PatienceSort(a []int) []int {
    piles := make([][]int, 0, 10)  // each pile will be a slice.
    for _, x := range(a) {
        i := bisectPilesRight(piles, x)
        if i < len(piles) {
            piles[i] = append(piles[i], x)
```

```
            } else {
                piles = append(piles, []int{x}) // make a new pile
            }
            // fmt.Println(piles)
        }

        h := PileHeap(piles) // Use piles as a heap
        // heap.Init(&h) is not need because piles are already sorted by top card
        n := len(a)
        sorted := make([]int, n)
        for i := 0; i < n; i++ {
            pile := heap.Pop(&h).([]int)
            top := len(pile) - 1
            sorted[i] = pile[top]
            if top > 0 {
                // Put pile minus the top card back in heap if it is not empty
                heap.Push(&h, pile[:top])
            }
        }
        return sorted
    }


    func main() {
        a := []int{2,6,3,1,5,9,2}
        fmt.Print(patienceSort(a))
    }
```

History [edit]

According to D. Aldous and P. Diaconis,[1] patience sorting was first recognized as an algorithm to compute the longest increasing subsequence length by Hammersley,[4] and by A.S.C. Ross and independently Robert W. Floyd as a sorting algorithm. Initial analysis was done by Mallows.[5]

Use [edit]

The Bazaar version control system uses the patience sorting algorithm for merge resolution.[6] The patience sorting algorithm can also be applied to process control. Within a series of measurements, the existence of a long increasing subsequence can be used as a trend marker. A 2002 article in SQL Server magazine includes a SQL implementation, in this context, of the patience sorting algorithm for the length of the longest increasing subsequence.[7]

References [edit]

1. ^ [a] [b] David Aldous and Persi Diaconis. Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem. *Bull. (new series) of the Amer. Math. Society, Volume 36*, number 4, pages 413–432, p.414
2. ^ [a] [b] Sergei Bespamyatnikh and Michael Segal. Enumerating Longest Increasing Subsequences and Patience Sorting. *Pacific Inst. for the Math. Sci. Preprints*, PIMS-99-3., pp.7–8
3. ^ http://rosettacode.org/wiki/Longest_increasing_subsequence#Clojure
4. ^ J.M. Hammersley. A few seedlings of research. In *Proc. Sixth Berkeley Symp. Math. Statist. and Probability, Volume 1*, pages 345–394. University of California Press, 1972. MR **53**:9457, p.362
5. ^ C.L. Mallows. Patience sorting. *Bull. Inst. Math. Appl.*, 9:216–224, 1973.
6. ^ http://revctrl.org/PreciseCodevilleMerge
7. ^ Kass, Steve (April 30, 2002). "Statistical Process Control". *SQL Server Pro*. Retrieved 23 April 2014.

The Wikibook *Algorithm implementation* has a page on the topic of: ***Patience sorting***

v · t · e	**Sorting algorithms**	[hide]
Theory	Computational complexity theory · Big O notation · Total order · Lists · Inplacement · Stability · Comparison sort · Adaptive sort · Sorting network · Integer sorting
Exchange sorts	Bubble sort · Cocktail sort · Odd–even sort · Comb sort · Gnome sort · Quicksort · Stooge sort · Bogosort
Selection sorts	Selection sort · Heapsort · Smoothsort · Cartesian tree sort · Tournament sort · Cycle sort
Insertion sorts	Insertion sort · Shellsort · Splaysort · Tree sort · Library sort · **Patience sorting**

Merge sorts	Merge sort · Cascade merge sort · Oscillating merge sort · Polyphase merge sort · Strand sort
Distribution sorts	American flag sort · Bead sort · Bucket sort · Burstsort · Counting sort · Pigeonhole sort · Proxmap sort · Radix sort · Flashsort
Concurrent sorts	Bitonic sorter · Batcher odd–even mergesort · Pairwise sorting network
Hybrid sorts	Block sort · Timsort · Introsort · Spreadsort · JSort
Other	Topological sorting · Pancake sorting · Spaghetti sort

Categories: Comparison sorts | Solitaire card games