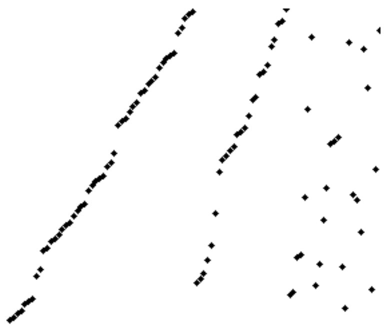
I, ME AND MYSELF !!!

THURSDAY, AUGUST 19, 2010

Merge Sort

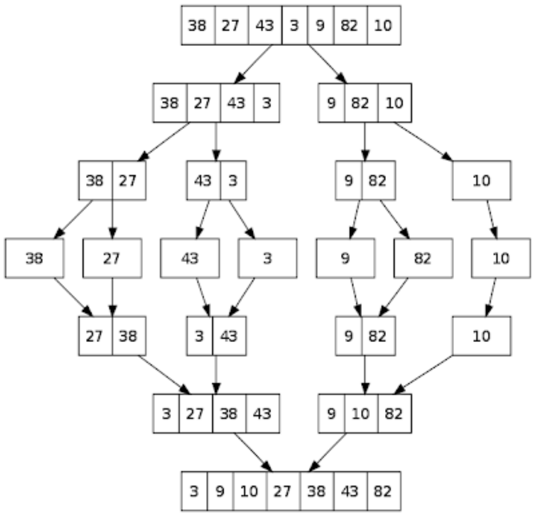
Introduction:

Merge-sort is based on the divide-and-conquer paradigm. The Merge-sort algorithm can be described in general terms as consisting of the following three steps:



1. **Divide Step:** If given array A has zero or one element, return S; it is already sorted. Otherwise, divide A into two arrays, A1 and A2, each containing about half of the elements of A.
2. **Recursive Step:** Recursively sort array A1 and A2.
3. **Conquer Step:** Combine the elements back in A by merging the sorted arrays A1 and A2 into a sorted sequence.

This diagram below shows the divide and conquer (or merging) steps stated above.



We can visualize Merge-sort by means of binary tree where each node of the tree represents a recursive call and each external nodes represent individual elements of given array A. Such a tree is called Merge-sort tree. The heart of the Merge-sort algorithm is conquer step, which merge two sorted sequences into a single sorted sequence. This simple but amazing algorithm and a straight forward C++ implemented is presented below, and some cool links are added in the "Reference" section at the end of the post.

Algorithm:

Input: Array A[p...r], indices p, q, r (p ≤ q < r).
Output: Array A[p...r] in ascending order

```
MERGE-SORT(A, p, r):
    if p < r
        then q := (r + p) / 2
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q+1, r)
        MERGE(A, p, q, r)

MERGE(A, p, q, r):
    n1 := q - p + 1
```

Followers (488)

Follow

SUBSCRIBE

Posts

Comments

BLOG HITS

BLOG ARCHIV

- ► 2015 (4)
- ► 2014 (6)
- ► 2013 (19)
- ► 2012 (14)
- ► 2011 (15)
- ▼ 2010 (33)
  - ► December
  - ► November
  - ► Septembe
  - ▼ August (3)
    - Merge Sort
    - Simple Tak
    - Binary Tree Height
  - ► June (2)
  - ► May (5)
  - ► March (5)
  - ► February (
  - ► January (3
- ► 2009 (27)

ABOUT ME

Vi
pr

```
    n2 := r - q
    create arrays L[1...N1+1] and R[1...N2+1]
    for i := 1 to N1
        do L[i] := A[p + i - 1]
    for j := 1 to N2
        do R[j] := A[q + j]
    L[N1 + 1] := INF
    R[N2 + 1] := INF
    i := 1
    j := 1
    for k := p to r
        do if L[i] <= R[j]
            then A[k] := L[i]
                i := i + 1
            else
                A[k] := R[j]
                j := j + 1
```

Follow this link to see a javascript demonstration that simulates the above algorithm.

Analysis:

In sorting n objects, merge sort has an average and worst-case performance of O(n log n). If the running time of merge sort for a list of length n is T(n), then the recurrence T(n) = 2T(n/2) + n follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists) and the closed form follows from the master theorem.

Simple proof:
The merge sort algorithm created a complete binary tree, which have d depth and at each level, a total of n elements.
So, 2^d ≈ n, which implies d ≈ lg n
Now the total numbers of operation in merge sort algorithm is:
n * 2d ≈ 2n lg n ≈ O(n lg n)

In the worst case, merge sort does an amount of comparisons equal to or slightly smaller than $(n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1)$, which is between (n lg n - n + 1) and (n lg n + n + O(lg n)).

In the worst case, merge sort does about 39% fewer comparisons than quicksort does in the average case. Merge sort always makes fewer comparisons than quicksort, except in extremely rare cases, when they tie, where merge sort's worst case is found simultaneously with quicksort's best case. In terms of moves, merge sort's worst case complexity is O(n log n) — the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case. Although, depending on the machine's memory architecture, quick sort can sometimes outperform merge sort, which is a very rare case.

Recursive implementations of merge sort make 2n - 1 method calls in the worst case, compared to quicksort's n, thus merge sort has roughly twice as much recursive overhead as quicksort. However, iterative, non-recursive, implementations of merge sort, avoiding method call overhead, are not difficult to code. Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in.

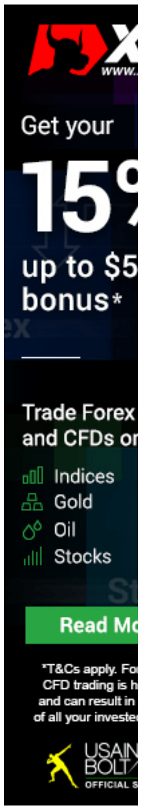
Merge sort as described here also has an often overlooked, but practically important, best-case property. If the input is already sorted, its complexity falls to O(n). Specifically, n-1 comparisons and zero moves are performed, which is the same as for simply running through the input, checking if it is pre-sorted.

Although heap sort has the same time bounds as merge sort, it requires only T(1) auxiliary space instead of merge sort's T(n), and is often faster in practical implementations. In case in-place sorting is necessary (Merge sort can be implemented as in-place algorithm, but the performance gain is not worth the complexity of the program, however, still merge sort runs in O(n lg n) time), heap sort is a better choice.

C++ Implementation:

Call Merge_Sort(A, start, end) to sort the closed range [start, end] of the array A.

```
void Merge(int A[], int p, int q, int r) {
    int i, j, k, n1 = q - p + 1, n2 = r - q;
    int L[n1], R[n2];
    for(i = 0; i < n1; i++)
        L[i] = A[p + i];
    for(j = 0; j < n2; j++)
        R[j] = A[q + j + 1];
    for(k = p, i = j = 0; k <= r; k++) {
        if(j >= n2 || (i < n1 && L[i] <= R[j])) A[k] = L[i++];
        else A[k] = R[j++];
    }
}
```



STACK OVERF

```
void Merge_Sort(int A[], int p, int r) {
    if(p < r) {
        int q = (p + r) / 2;
        Merge_Sort(A, p, q);
        Merge_Sort(A, q+1, r);
        Merge(A, p, q, r);
    }
}
```

For implementations in other languages, this page contains the implementation of merge sort procedure in almost all the living languages of the world: http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort

Reference:

- Introduction To Algorithms (CLRS, MIT Press, 2nd Edition): Chapter 2, Section 2.3.1 - The divide-and-conquer approach.
- http://en.wikipedia.org/wiki/Merge_sort
- http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/mergen.htm
- http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm

Thanks for reading.

Posted by Zobayer Hasan at 8:31 PM

2 comments:

Anonymous October 22, 2012 at 6:52 PM

The code for C++ is working great.

Thank you.

Reply

Replies

Zobayer Hasan October 22, 2012 at 7:21 PM

You may also like to check the threaded version I implemented here. It can be divided into computers over the network if you can write the network API parts yourself.

Reply

Enter your comment...

Comment as: rayhan_ahmed ▼ Sign out

Publish Preview Notify me