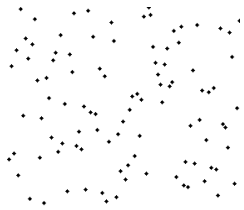
Mergesort algorithm



Consider the sorting problem discussed in another chapter. Namely, consider the problem of permuting the elements of an array `v[0..n-1]` to put them in increasing order, i.e., rearrange the elements so that `v[0]` ≤ . . . ≤ `v[n-1]`. The previous chapter analysed some basic algorithms for the problem. Those algorithm are quadratic, i.e., they consume an amount of time proportional to n^2.

The present chapter looks at a more sophisticated and much faster algorithm that uses the divide-and-conquer strategy. The basic idea is simple: if the first half of the array is already increasing and the second half is also increasing, then the two halves can be quickly merged so that the whole array is increasing.

Table of contents:

Merging sorted arrays

Before we can deal with our sorting problem, we must consider the following auxiliary *merging* problem: given increasing arrays `v[p..q-1]` and `v[q..r-1]`, rearrange the array `v[p..r-1]` in increasing order.

p						q-1	q			r-1
111	333	555	555	777	999	999	222	444	777	888

It would be easy to solve the problem in an amount of time proportional to the square of the size of the whole array `v[p..r-1]`: just ignore that the two "halves" are already sorted and use one of the basic sorting algorithms. But we can do much better. In order to do it, we need a workspace, say `w[0..r-p-1]`, of the same type and size as the array `v[p..r-1]`.

```
// This function receives increasing arrays
// v[p..q-1] and v[q..r-1] and rearranges
// v[p..r-1] in increasing order.

static void
merge (int p, int q, int r, int v[])
```

```
{
   int *w;  // 1
   w = malloc ((r-p) * sizeof (int));  // 2
   int i = p, j = q;  // 3
   int k = 0;  // 4
   while (i < q && j < r) {  // 5
      if (v[i] <= v[j])  w[k++] = v[i++];  // 6
      else  w[k++] = v[j++];  // 7
   }  // 8
   while (i < q)  w[k++] = v[i++];  // 9
   while (j < r)  w[k++] = v[j++];  // 10
   for (i = p; i < r; ++i)  v[i] = w[i-p]; // 11
   free (w);  // 12
}
```

0:00 / 1:58

The keyword `static` indicates that the function `merge` has an auxiliary nature and will not be called directly by the user of the sorting algorithm.

Performance. Function `merge` consists essentially of *moving* the elements of array `v` from one place to another (first from `v` to `w` and then back from `w` to `v`). The function executes

$$2n$$

of these moves, where `n` is the size of the array `v[p..r-1]`, i.e., `n = r-p`. The time spent by `merge` is proportional to the number of moves. Therefore, the time consumption of the function is proportional a `n`. In other words, `merge` is linear.

Exercises 1

1. Write a function that receives disjoint arrays `x[0..m-1]` and `y[0..n-1]`, both in increasing order, and produces an array `z[0..m+n-1]` that contains the result of merging the two given arrays. Write an iterative and a recursive versions.
2. Is the function `merge` correct when the array `v[p..q-1]` is empty (i.e., when `p == q`)? Is the function correct when the array `v[q..r-1]` is empty?
3. Replace lines `9` to `11` of `merge` by the two following lines. Does the function remain correct?

   ```
   while (i < q)  w[k++] = v[i++];
   for (i = p; i < j; ++i)  v[i] = w[i-p];
   ```

4. Criticize the effect of replacing lines `5` to `8` of the `merge` code by the following lines.

   ```
   while (i < q && j < r) {
      if (v[i] <= v[j])  w[k++] = v[i++];
      if (v[i] > v[j])  w[k++] = v[j++]; }
   ```

5. Replace lines `3` to `10` of the `merge` code by the lines below. Does the function remain correct?

   ```
   i = p; j = q;
   for (k = 0; k < r-p; ++k) {
      if (j >= r || (i < q && v[i] <= v[j]))
         w[k] = v[i++];
      else
         w[k] = v[j++]; }
   ```

6. Replace lines `5` to `10` of `merge` by the lines below. Does the function remain correct?

   ```
   while (k < r-p) {
      while (i < q && v[i] <= v[j])
   ```

```
      w[k++] = v[i++];
   while (j < r && v[j] <= v[i])
      w[k++] = v[j++]; }
```

7. Invariants. What are the invariants of the first "`while`" (lines `5` to `8`) in the `merge` function?

8. Show that the time consumption of function `merge` is not proportional to the number of comparisons between elements of the array.

9. The following variant of the `merge` function does the merging *in-place*, i.e., without any auxiliary array. (It inserts each element of `v[q..r-1]` into `v[p..q-1]` as in the Insertionsort algorithm.) Criticize the variant.

```
while (q < r) {
   int x = v[q], int i;
   for (i = q-1; i >= p && v[i] > x; --i)
      v[i+1] = v[i];
   v[i+1] = x;
   q++; }
```

10. Is the following solution of the merging problem correct? What are the invariants of the "`while`"? (Notice that the function operates *in-place*, that is, without an auxiliary array.) How much time does the function consume?

```
int i, k, x;
i = p;
while (i < q && q < r) {
   if (v[i] >= v[q]) {
      x = v[q];
      for (k = q - 1; k >= i; --k)
         v[k+1] = v[k];
      v[i] = x;
      ++q; }
   ++i; }
```

11. In-place merging challenge. Invent a function that is as fast as `merge` and solves the merging problem *in-place*, i.e., without an auxiliary array.

12. A merging algorithm is *stable* if does not change the relative positions of same-value elements. Is the `merge` function discussed above stable? What if the comparison "`v[i] <= v[j]`" is replaced by "`v[i] < v[j]`"?

13. Merging of linked lists. Let's say, for the purpose of this exercise, that an *i-list* is a (headless) linked list that contains an increasing sequence of integers. Write a function to produce an i-list by merging two given i-lists. Your function should not allocate new cells, but recycle the cells of the given i-lists.

14. Union of linked lists. Let's say, for the purpose of this exercise, that an *si-list* is a (headless) linked list that contains a strictly increasing sequence of integers. (Hence, an si-list represents a *set* of integers.) Write a function to do the *union* of two si-lists. The resulting list must be an si-list and must be built from the cells of the two given lists.

Merging with sentinels

Sedgewick writes the merging algorithm in the following clever way. First, copy the array `v[p..q-1]` to the workspace `w[0..q-p-1]`; then, copy `v[q..r-1]` to the workspace `w[q-p..r-p-1]` *in reverse order*. Now, the left "half" of `w` serves as a sentinel for the right "half", and vice-versa, during the merging process. As a result, there is no need to check the boundary conditions `i < q-p` and `j ≥ q-p` at every iteration.

```
// This function receives increasing arrays
// v[p..q-1] and v[q..r-1] and rearranges
// v[p..r-1] in increasing order.

static void
s_merge (int p, int q, int r, int v[])
{
   int i, j, *w;
   w = malloc ((r-p) * sizeof (int));

   for (i = p; i < q; ++i) w[i-p] = v[i];
   for (j = q; j < r; ++j) w[r-p+q-j-1] = v[j];
```

```
   i = 0; j = r-p-1;
   for (int k = p; k < r; ++k)
      if (w[i] <= w[j]) v[k] = w[i++];
      else v[k] = w[j--];
   free (w);
}
```

Just as the previous version, this one takes time proportional to the size of the array `v[p..r-1]`.

Exercises 2

1. Discuss the following variation of `s_merge`:

   ```
   for (i = 0, k = p; k < q; ++i, ++k)
      w[i] = v[k];
   for (j = r-p-1, k = q; k < r; --j, ++k)
      w[j] = v[k];
   i = 0; j = r-p-1;
   for (k = p; k < r; ++k)
      if (w[i] <= w[j]) v[k] = w[i++];
      else v[k] = w[j--];
   ```

2. [Sedgewick] Show that function `s_merge` is not stable. How can we modify the code so that the function becomes stable?

Mergesort algorithm

The Mergesort algorithm uses a divide-and-conquer strategy to sort the given array. The divide phase is simple: just break the array in half. The conquer phase is implemented by the `merge` function discussed above.

The recursive function shown next rearranges the array `v[p..r-1]` in increasing order. The basis of the recursion is the set of instances where `p ≥ r-1`; for these instances, the array has at most 1 element and therefore nothing needs to be done to put it in increasing order.

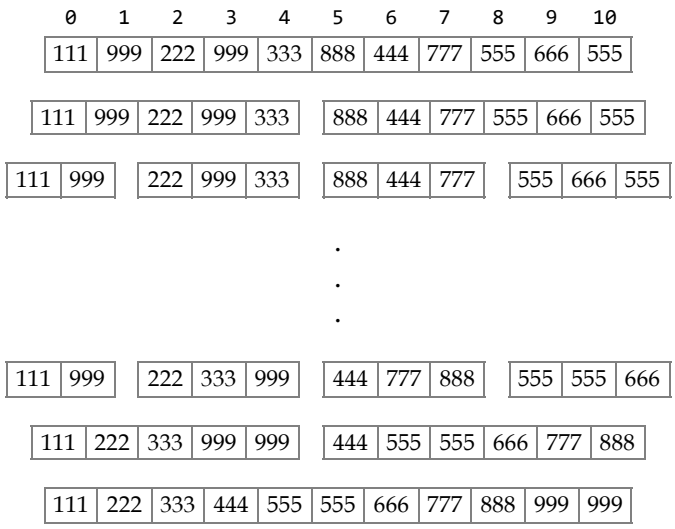
```
// The function mergesort rearranges the
// array v[p..r-1] in increasing order.

void
mergesort (int p, int r, int v[])
{
   if (p < r-1) {              // 1
      int q = (p + r)/2;       // 2
      mergesort (p, q, v);     // 3
      mergesort (q, r, v);     // 4
      merge (p, q, r, v);      // 5
   }
}
```

0:00 / 1:35

(Notes: 1. You can replace the call to `merge` in line `5` by a call to `s_merge`, since these two functions are equivalent. 2. The result of the division by `2` in the expression `(p+r)/2` is automatically truncated. For example, `(3+6)/2` is `4`.)

If `p < r-1`, the instance `v[p..r-1]` of the problem is reduced to the pair of instances `v[p..q-1]` and `v[q..r-1]`. These two instances are strictly smaller than the original instance, since `q < r` and `q > p` (check it!) at the end of

line 2. Hence, by induction hypothesis, the array `v[p..q-1]` will be in increasing order at the end of line 3, and the array `v[q..r-1]` will be in increasing order at the end of line 4. Now, at the end of line 5, the array `v[p..r-1]` will be in increasing order thanks to the merge operation. This discussion proves that the function `mergesort` is correct.



To rearranged in increasing order an array `v[0..n-1]`, as the original formulation of the problem required, all we need to do is execute `mergesort (0, n, v)`.

Exercises 3

1. Show that `p < q < r` at the end of line 2 of `mergesort`.
2. What happens if we replace "`(p+r)/2`" by "`(p+r-1)/2`" in the code of the `mergesort` function? What happens if we replace "`(p+r)/2`" by "`(p+r+1)/2`"?
3. Call the `mergesort` function on an array indexed by `1..4`. You will have the following sequence of calls (note the indentation):

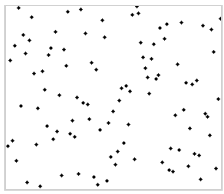
```
mergesort (1,5,v)
    mergesort (1,3,v)
        mergesort (1,2,v)
        mergesort (2,3,v)
    mergesort (3,5,v)
        mergesort (3,4,v)
        mergesort (4,5,v)
```

   Repeat this exercise with an array indexed by `1..5`.

4. TRICK QUESTION. What are the invariants of the `mergesort` function?
5. CORRECTNESS CHECK. Write a program to test, experimentally, the correctness of your implementation of the Mergesort algorithm. (See the analogous exercise for Insertionsort.)
6. Is `mergesort` a stable sorting function?
7. OVERFLOW. If the size of the array is close to `INT_MAX`, the execution of `mergesort` can derail on line 2 due to an arithmetic overflow. How could this be avoided?

Animations of Mergesort

The animation at right (copied from Wikipedia) shows the sorting of an array `v[0..99]` that contains a random permutation of `0..99`. (See a slower version of the animation.) Each element `v[i]` is represented by the point with coordinates `(i, v[i])`.



There are many other animations of Mergesort on the Web. Here is a sample:

- Comparison Sorting Algorithms, by David Galles (University of San Francisco),
- Sorting Algorithms Animations, on Toptal,
- animation of 15 sorting algorithms in 6 minutes, by Timo Bingmann, on YouTube,
- Merge-sort with Transylvanian-saxon folk dance, from the Sapientia University in Romania.

Exercises 4

1. The following function promises to rearrange `v[p..r-1]` in increasing order. Is the function correct?

```
void mergesort1 (int p, int r, int v[]) {
    if (p < r-1) {
       int q = (p + r) / 2;
       mergesort1 (p, q, v);
       mergesort1 (q, r, v);
       merge (p, q+1, r, v); } }
```

2. The following function promises to rearrange `v[p..r-1]` in increasing order. Is the function correct?

```
void mergesort2 (int p, int r, int v[]) {
    if (p < r) {
       int q = (p + r) / 2;
       mergesort2 (p, q, v);
       mergesort2 (q, r, v);
       merge (p, q, r, v); } }
```

3. Is the following function correct? It promises to rearrange `v[p..r-1]` in increasing order.

```
void mergesort3 (int p, int r, int v[]) {
    if (p < r-1) {
       int q = (p + r - 1) / 2;
       mergesort3 (p, q, v);
       mergesort3 (q, r, v);
       merge (p, q, r, v); } }
```

4. Does the function below rearrange `v[p..r-1]` in increasing order? What if we replace "`(p+r)/2`" by "`(p+r+1)/2`"?

```
void mergesort4 (int p, int r, int v[]) {
    if (p < r-1) {
       int q = (p + r) / 2;
       mergesort4 (p, q-1, v);
       mergesort4 (q-1, r, v);
       merge (p, q-1, r, v); } }
```

5. Does the function below rearrange `v[p..r-1]` in increasing order?

```
void mergesort5 (int p, int r, int v[]) {
   if (p < r-1) {
      q = r - 2;
      mergesort5 (p, q, v);
      if (v[r-2] > v[r-1]) {
         int t = v[r-2];
         v[r-2] = v[r-1];
         v[r-1] = t; }
      merge (p, q, r, v); } }
```

6. Does this function rearrange `v[p..r-1]` in increasing order?

```
void mergesort6 (int p, int r, int v[]) {
   if (p < r-1) {
      q = r - 1;
      mergesort6 (p, q, v);
      merge (p, q, r, v); } }
```

7. Suppose your library has a function `mrg(p,q,r,v)` that rearranges the array `v[p..r-1]` is increasing order assuming that `v[p..q]` and `v[q+1..r-1]` are already in increasing order. Use `mrg` to implement the Mergesort algorithm.

8. Suppose your library has a function `mrg(v,p,q,r)` that receives an array `v` such that `v[p..q-1]` and `v[q..r-1]` are in increasing order and rearranges the array so that `v[p..r-1]` is in increasing order. (What is the smallest value of `q` that `mrg` must accept? What is the greatest value?) Use `mrg` to write a function `mrgsrt(v,p,r)` that will rearrange an array `v[p..r]` in increasing order.

Performance of the Mergesort algorithm

Submit an array `v[0..n-1]` to the `mergesort` function. The size of the array is reduced by half in each step of the recursion. In the first round, the original instance of the problem is reduced to the instances `v[0..n/2-1]` and `v[n/2..n-1]`. In the second round, we have four instances:

`v[0..n/4-1]`, `v[n/4..n/2-1]`, `v[n/2..3n/4-1]` and `v[3n/4..n-1]`.

And so on, until, in the last round, each instance has at most 1 element. The total number of rounds is approximately $\log n$ (therefore also approximately $\lg(n)$).

In each round, the function `merge` moves $2n$ elements of the array `v[0..n-1]` (why?). Hence, the total number of moves executed to sort `v[0..n-1]` is approximately

$$2n \log n .$$

It is easy to see that the *time* consumed by `mergesort` is proportional to the total number of moves, and therefore proportional to

$$n \log n .$$

Hence, the algorithm is linearithmic. The number $n \log n$ grows much slower than n^2 and only a little faster than n. If an array of size N requires T units of time, an array of size $2N$ will require less than $2.2\,T$ units of time, provided N is greater than 2^{10}. Likewise, an array of size $4N$ will require less than $4.4\,T$ units of time, provided N is greater than 2^{20}. (Check the math!)

The time consumption of Mergesort is proportional to $n \log n$ while that of the basic algorithms is proportional to n^2. But the factor of proportionality is larger in the case of Mergesort, since the code is more complex. Hence, Mergesort only becomes really faster than the basic algorithms when n is sufficiently large. (This is a very common phenomenon: sophisticated algorithms are typically slower than simple algorithms when the amount of data is small.)

Exercises 5

1. How does the time consumption of the following code fragment depend on `n`?

```
int c = 1;
for (int i = 0; i < n; i *= 2)
   for (int 0 = 1; j < n; ++j)
      c += 1;
```

2. Cutoff version. Write a version of Mergesort with a *cutoff* for small arrays: when the size of the array to be sorted falls below M, switch to the Insertionsort algorithm. You can take M between 10 and 20, say. (This cutoff trick is used in practice because Insertionsort is faster than "pure" Mergesort when the array is small.)

3. Excessive allocation/deallocation. The function `mergesort` given above calls the functions `malloc` and `free` many times (these calls happen within `merge`). Write a version of `mergesort` that incorporates the code of the `merge` function and calls `malloc` only once.

4. Challenge: in-place Mergesort. Invent an implementation of Mergesort that will do the sorting *in-place*, that is, without an auxiliary array. Your implementations must be linearithmic.

5. Decreasing order. Write a version of the Mergesort algorithm to rearrange an array `v[p..r-1]` in decreasing order.

6. The following recursive function promises to find the value of a largest element of the (not necessarily sorted) array `v[p..r]`. Of course the problem only makes sense if `p ≤ r`.

```
int max (int p, int r, int v[]) {
   if (p == r) return v[r];
   else {
      int q = (p + r)/2;
      int x = max (p, q, v);
      iny y = max (q+1, r, v);
      if (x >= y) return x;
      else return y; } }
```

   Is the function correct? Is it faster than the obvious iterative function? How many times does the function call itself? Suppose that `p` and `r` are `0` and `6` respectively and show the (duly indented) sequence of calls of `max`.

7. Performance test. Write a program to time your implementation of the Mergesort algorithm. (See the analogous exercise for Insertionsort. For Mergesort, you may run tests for a longer sequence of values of n, perhaps 2^8, 2^9, ... , 2^{29}, 2^{30}.)

Iterative version of Mergesort

The Mergesort algorithm can be implemented in iterative style. In each iteration, we merge two consecutive blocks of `b` elements each: the first block with the second, the third with the fourth, and so on. Variable `b` assumes the values `1, 2, 4, 8, . . .`

```
// This function rearranges the array
// v[0..n-1] in increasing order.

void
imergesort (int n, int v[])
{
   int b = 1;
   while (b < n) {
      int p = 0;
      while (p + b < n) {
         int r = p + 2*b;
         if (r > n) r = n;
         merge (p, p+b, r, v);
         p = p + 2*b;
      }
      b = 2*b;
   }
}
```

The figure illustrates the iteration in which `b` is `2`:

0				p		p+b		p+2b		n-1
111	999	222	999	333	888	444	777	555	666	555

There are many interesting animations and visualizations of the iterative version of Mergesort:

- Animation of Mergesort produced by Mike Bostock. [Suggested by Yoshiharu Kohayakawa.] The array is represented by a "broom". Each hair of the "broom" is an element of the array and the inclination of the

hair is the value of the element. Here are some standalone presentations of the animation:

- Mergesort I. (See the code.)
- Mergesort II. The segments being merged are highlighted. (See the code.)
- Mergesort III. A static visualization. (See the code.)

- Mergesort IV, by Mike Bostock. An animation/visualization using colored twisting strings. Reload to see variations of the data. (See the code.)

Exercises 6

1. INVARIANTS. What are the invariants of the outer "`while`" in `imergesort`? What are the invariants of the inner "`while`"?

2. INCREASING SEGMENTS. Function `imergesort` begins by breaking the original array into segments of length 1. Why not begin with maximal increasing segments? Example: the maximal increasing segments of the array 1 2 3 0 2 4 6 4 5 6 7 8 9 are 1 2 3, 0 2 4 6 and 4 5 6 7 8 9. Explore this ideia.

Exercises 7

1. LINKED LISTS. Write a version of the Mergesort algorithm that will rearrange a linked list in increasing order. Your function must not allocate new cells in memory. Write a recursive and an iterative versions.

2. NUMBER OF INVERSIONS. The *number of inversions* of an array `v[0..n-1]` is the number of ordered pairs `(i,j)` such that `0 ≤ i < j < n` and `v[i] > v[j]`. Write a function to compute the number of inversions of a given array. The time consumption of your function must be proportional to `n` log `n` in the worst case.

3. KENDALL TAU DISTANCE. Suppose you are given two permutations, say `x[0..n-1]` and `y[0..n-1]`, of the same set of numbers. The *tau distance* between `x` and `y` is the number of pairs of elements of the set that are in a different order in `x` and `y`, that is, the cardinalidade of the set $X - Y$ where X is the set of all the pairs `(x[i],x[j])` such that `i < j` and Y is the set of all the pairs `(y[i],y[j])` such that `i < j`. (The definition is not asymmetric as it seems since the sets $X - Y$ and $Y - X$ have the same cardinalidade.) Write an efficient function to compute the tau distance between `x` and `y`.

See the *Merge sort* entry in Wikipedia.

See the *Timsort* algorithm, based on ideias similar to Mergesort.

Updated 2019-03-01
https://www.ime.usp.br/~pf/algorithms/

CS-IME-USP