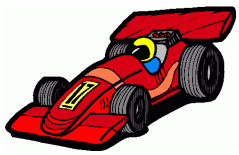
# Quicksort



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 of the array so that `v[0]` ≤ . . . ≤ `v[n-1]`.

That previous chapter analysed some simple algorithms for the problem. The present chapter looks at a more sophisticated algorithm — the Quicksort — invented by C.A.R. Hoare in 1962. In some rare instances, Quicksort is as slow as the simple algorithms, but in general it is much faster. More precisely, the algorithm is linearithmic on average but only quadratic in the worst case.

Here is the basic ideia of the algorithm: if the left half of the array contains the small elements and the right half contains the large elements, then all we have to do is rearrange each half in increasing order independently.

We shall use two shorthands to discuss the algorithm. The expression "`v[i..k]` ≤ `x`" will be used as a shorthand for "`v[j]` ≤ `x` for all `j` in the set of indices `i..k`". The expression "`v[i..k]` ≤ `v[p..r]`" will be interpreted as "`v[j]` ≤ `v[q]` for all `j` in the set `i..k` and all `q` in the set `p..r`".

Table of contents:



## The partition problem

The core of the Quicksort algorithm is a *partition subproblem*, that we shall formulate rather vaguely on purpose:

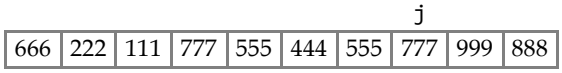
rearrange an array `v[p..r]` so that all the small elements fall into the "left" part of the array and all the large elements fall into the "right" part.

The begin solving this problem, we must choose a *pivot*, say `c`. The elements of the array that are greater than `c` will be considered large and the remaining ones will be considered small. It is important to choose `c` so that each part of the rearranged array is strictly smaller than the whole array. The challenge is to solve the partition subproblem quickly and without resorting to an auxiliary array for workspace.

There are many concrete formulations of the partition problem. Here is the first one: rearrange `v[p..r]` so that `v[p..j] ≤ v[j+1..r]` for some `j` in `p..r-1`. (In this formulation, the pivot is not explicit.) Here is a second formulation: rearrange `v[p..r]` so that

`v[p..j-1] ≤ v[j] < v[j+1..r]`

for some `j` in `p..r`. (Here, `v[j]` is the pivot.) An example:




In this chapter, we shall use the second formulation of the partition problem.

## Exercises 1

1. Positive and negative. Write a function to rearrange an array `v[p..r]` of integers so that `v[p..j-1] ≤ 0` and `v[j..r] > 0` for some `j` in `p..r+1`. Does it make sense to require that `j` be in `p..r`? Try to write a *fast* function that does not use an auxiliary array. Repeat the exercise after changing "`v[j..r] > 0`" into "`v[j..r] ≥ 0`". Does it make sense to require that `v[j] == 0`?

2. Linked list. Suppose you are given a linked list that stores strictly positive integer numbers. Write a function to divide the list in two: the first containing the even numbers and the second containing the odd ones.

3. Criticize the following tentative solution of the partition problem:

```
int prttn (int v[], int p, int r) {
   int j = r;
   for (int i = r-1; i >= p; i--)
      if (v[i] > v[r]) {
         int t = v[r]; v[r] = v[i]; v[i] = t;
         j = i; }
   return j; }
```

4. Criticize the following solution of the partition problem:

```
int prttn (int v[], int p, int r) {
   int w[1000], i = p, j = r, c = v[r];
   for (int k = p; k < r; ++k)
      if (v[k] <= c) w[i++] = v[k];
      else w[j--] = v[k];
   // now i == j
   w[j] = c;
   for (int k = p; k <= r; ++k) v[k] = w[k];
   return j; }
```

5. A programmer claims that the following function solves the first formulation of the partition problem, that is, it rearranges any array `v[p..r]` (with `p < r`) so that `v[p..i] ≤ v[i+1..r]` for some `i` in `p..r-1`.

```
int prttn (int v[], int p, int r) {
   int i = p, j = r;
   int q = (p + r)/2;
   do {
```

```
      while (v[i] < v[q]) ++i;
      while (v[j] > v[q]) --j;
      if (i <= j) {
         int t = v[i], v[i] = v[j], v[j] = t;
         ++i, --j; }
   } while (i <= j);
   return i; }
```

Show an example in which this function does not produce the promised result. What if we replace "`return i`" by "`return i-1`"? Is it possible to make a few changes to the code to make the function work as promised?

6. ★ Let's say that an array `v[p..r]` is *partitioned* if there exists `j` in `p..r` such that `v[p..j-1]` ≤ `v[j]` < `v[j+1..r]`. Write an algorithm to decide whether an array `v[p..r]` is partitioned. In the "yes" case, your algorithm should return the index `j`.

# A partition algorithm

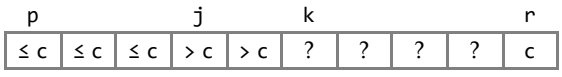
Here is how the book by Cormen, Leiserson, Rivest and Stein solves the partition problem. (The algorithm is attributed to N. Lomuto.) The function solves the second formulation of the partition problem:

```
// Rearranges the elements of the array
// v[p..r], with p <= r, and returns j
// in p..r such that v[p..j-1] <= v[j] <
// v[j+1..r].

static int
partition (int v[], int p, int r) {
   int c = v[r]; // pivot
   int t, j = p;
   for (int k = p; /*A*/ k < r; ++k)
      if (v[k] <= c) {
         t = v[j], v[j] = v[k], v[k] = t;
         ++j;
      }
   t = v[j], v[j] = v[r], v[r] = t;
   return j;
}
```

(The keyword `static` indicates that `partition` is an auxiliary function and prevents the user of Quicksort from calling it directly.)

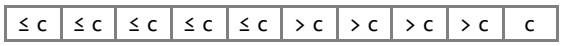
To show that the function `partition` is correct, just check that at the start of each iteration — that is, every time the execution transits through point *A* — we have the following configuration:



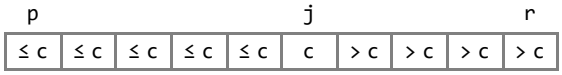

(Note that we can have `j == p` as well as `k == j`, that is, the left or the right half of the array can be empty.) More precisely, the following invariants hold: at the start of each iteration,

1. `p` ≤ `j` ≤ `k`,
2. `v[p..j-1]` ≤ `c`,
3. `v[j..k-1]` > `c`, and
4. `v[p..r]` is a permutation of the original array.

At the last transit through point *A*, we shall have `k == r` and therefore the following configuration:




Hence, when the execution of the function comes to an end, we shall have `p ≤ j ≤ r` and `v[p..j-1] ≤ v[j] < v[j+1..r]`, as promised.




We would like `j` to be halfway between `p` and `r`, but we must accept the extreme cases in which `j` is `p` or `r`.

**Performance.** How much time does the `partition` function take? The time consumption is proportional to the number of comparisons between elements of the array and this number is proportional to the size `r - p + 1` of the array. The function is, therefore, linear.

## Exercises 2

1. Run `partition(v, 0, 15)` assuming that `v[0..15]` is the array `33 22 55 33 44 22 99 66 55 11 88 77 33 88 66 66`.
2. Extreme case. What is the outcome of `partition(v, p, r)` when `p == r`?
3. Extreme cases. What is the outcome of `partition(v, p, r)` when the elements of `v[p..r]` are all equal? What if `v[p..r]` is increasing? What if `v[p..r]` is decreasing? What if each element of `v[p..r]` has one of two possible values?
4. Invariants. Prove that the invariants of `partition` function stated above are correct.
5. Recursive version. Write a recursive version of the function `partition`.
6. First element as pivot. Rewrite the code of `partition` taking `v[p]` to be the pivot.
7. Does the function `partition` produce a stable rearrangement of the array? In other words,, does it preserve the relative order of elements of same value?
8. Gries solution (from *The Science of Programming* by D. Gries). Analyse and discuss the following implementation of the partition algorithm. Show that this implementation is equivalent to Lomuto's `partition`. [Solution: ./solutions/quick8.html]

```
int partition_G (int v[], int p, int r) {
   int c = v[p], i = p+1, j = r;
   while (true) {
      while (i <= r && v[i] <= c) ++i;
      while (c < v[j]) --j;
      if (i >= j) break;
      int t = v[i], v[i] = v[j], v[j] = t;
      ++i; --j; }
   v[p] = v[j], v[j] = c;
   return j; }
```

9. Criticize the following variant of the function `partition_G`:

```
int prtG (int v[], int p, int r) {
   int c = v[p], i = p+1, j = r;
   while (i <= j) {
      if (v[i] <= c) ++i;
      else {
         int t = v[i], v[i] = v[j], v[j] = t;
         --j; } }
   v[p] = v[j], v[j] = c;
   return j; }
```

10. Roberts. Criticize the following solution of the partition problem:

```
int partition_R (int v[], int p, int r) {
   int c = v[p], i = p+1, j = r;
   while (1) {
      while (i < j && c < v[j]) --j;
      while (i < j && v[i] <= c) ++i;
      if (i == j) break;
      int t = v[i], v[i] = v[j], v[j] = t; }
   if (v[j] > c) return p;
   v[p] = v[j], v[j] = c;
   return j; }
```

11. CHALLENGE. Write a solution of the partition problem that returns an index `j` such that `(r-p)/10 ≤ j-p ≤ 9(r-p)/10`.

12. An array is *binary* if each of its elements is `0` or `1`. Write a function to rearrange a binary array `v[0..n-1]` in increasing order. Your function should be linear.

## Basic Quicksort

Now that the partition problem is solved, we can take care of Quicksort proper. The algorithm uses the divide and conquer strategy and looks like an upended Mergesort:

```
// This function rearranges an array
// v[p..r] in increasing order.

void
quicksort (int v[], int p, int r)
{
   if (p < r) {  // 1
      int j = partition (v, p, r);  // 2
      quicksort (v, p, j-1);  // 3
      quicksort (v, j+1, r);  // 4
   }
}
```

If `p ≥ r` (this is the basis of the recursion), there is nothing to do. If `p < r`, the code reduces the instance `v[p..r]` of the problem to the pair of instances `v[p..j-1]` and `v[j+1..r]`. Since `p ≤ j ≤ r`, these two instances are strictly smaller than the original instance. Hence, by induction hypothesis, `v[p..j-1]` will be in increasing order at the end of line `3` and `v[j+1..r]` will be in increasing order at the end of line `4`. Since `v[p..j-1] ≤ v[j] < v[j+1..r]` due to the `partition` function, the whole array will be in increasing order at end of line `4`. This argument proves the function correct.

We can delete the second recursive call (line `4`) and replace the "`if`" by a "`while`". The resulting code is equivalent to `quicksort`:

```
void qcksrt (int v[], int p, int r) {
   while (p < r) {
      int j = partition (v, p, r);
      qcksrt (v, p, j-1);
      p = j + 1;
   }
}
```

## Exercises 3

1. What happens if we replace "`p < r`" by "`p <= r`" on line `1` of `quicksort`? What happens if we replace "`p < r`" by "`p != r`" on line `1`?

2. What happens if we replace "`j-1`" by "`j`" on line `3` of `quicksort`? What happens if we replace "`j+1`" by "`j`" on line `4`?

3. TRICK QUESTION. What are the invariants of function `quicksort`?

4. Let `v[1..4]` be the array `77 55 33 99`. If you run `quicksort (v, p, r)`, you will see the following sequence of calls:

```
quicksort (v,1,4)
    quicksort (v,1,2)
        quicksort (v,1,0)
        quicksort (v,2,2)
    quicksort (v,4,4)
```

   Repeat the exercise for the array `v[1..6]` = `55 44 22 11 66 33` .

5. Write a function with prototype `quick_sort (int v[], int n)` to rearrange an array `v[0..n-1]` (watch the indices!) in increasing order. (Hint: call `quicksort` in the appropriate way.)

6. Does function `quicksort` implement a stable sorting?

7. Discuss the following implementation of the Quicksort algorithm. Assume `p < r`.

```
void qsrt (int v[], int p, int r) {
   int j = partition (v, p, r);
   if (p < j-1) qsrt (v, p, j-1);
   if (j+1 < r) qsrt (v, j+1, r);
}
```

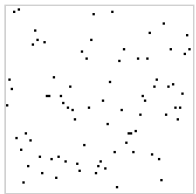
8. ★ ALTERNATIVE FORMULATION OF PARTITION. Suppose `prtt2` is a function that solves the first formulation of the partition problem (that is, it rearranges `v[p..r]` and returns `j` such that $v[p..j] \le v[j+1..r]$.) Write a version of the Quicksort algorithm that uses `prtt2`. What restrictions must be imposed on `j`?

9. ★ ITERATIVE VERSION. Write a nonrecursive implementation of the Quicksort algorithm. [Solution: ./solutions/quick-it.html]

10. ★ RANDOMIZED QUICKSORT. Write a randomized version of the Quicksort algorithm.

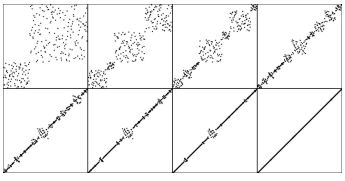
11. CORRECTNESS CHECK. Write a program to test, experimentally, the correctness of your implementation of the Quicksort algorithm. (See the analogous exercise for Insertionsort.)

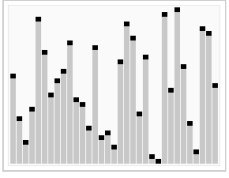
# Animations of Quicksort



The animation at right (copied from Simon Waldherr / Golang Sorting Visualization) shows Quicksort running on an array `v[0..79]` of positive numbers. Each element `v[i]` of the array is represented by the point `(i, v[i])`.

The first figure below (copied from Wikimedia) shows Quicksort running on a permutation `v[0..249]` of the set `0..249`. The figure shows snapshots of the array at 8 moments during the execution of the algorithm. The second figure (copied from Wikipedia) shows Quicksort running on a permutation `v[0..32]` of the set `0..32`.





There are many other animations of Quicksort. Here are some examples:

- Animation of Quicksort, by C. Parsons, on YouTube.
- Animation of Quicksort 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:
  - Quicksort. The pivot is highlighted in red. (See the code.)
  - Quicksort IV. Shows the active partition in black and the active pivot in red. The algorithm uses the value of the middle element of the array as pivot, i.e., it interchanges `v[(p+r)/2]` and `v[r]` before doing the partition. (See the code.)
  - Quicksort II. A static visualization of the iterative version. Each row corresponds to the state of the array prior to recursing into each partition. The red lines represent the pivots. Reload to see variations of the data. (See the code.)
  - Quicksort V. A static visualization using colored twisting strings. Each row represents the state of the array after a single swap operation. Reload to change the data. (See the code.)
  - Quicksort VI. An animated version of Quicksort V. (See the code.)
- Quick-sort with Hungarian folk dance, from the Sapientia University in Romania.
- Animation of 15 sorting algorithms, by Timo Bingmann, on YouTube.
- Comparison Sorting Algorithms, by David Galles (University of San Francisco).
- Sorting Algorithms Animations on Toptal.

## Performance of basic Quicksort

The function `quicksort` consumes time proportional to the number of comparisons between elements of the array. If the index `j` returned by `partition` is always more or less halfway between `p` and `r`, the number of comparisons will be approximately $n \log n$, where $n$ is the size, `r-p+1`, of the array. On the other hand, if the array is already sorted or nearly sorted, the number of comparisons is approximately

$$n^2.$$

Therefore, the worst case of `quicksort` is no better than that of the basic algorithms. Fortunately, the worst case is very rare: *on average*, the time consumption of `quicksort` is proportional to

$$n \log n.$$

(But the proof of this statement is not easy.) Therefore, the algorithm is linearithmic on average but quadratic in the worst case.

## Exercises 4

1. CUTOFF VERSION. Write a version of Quicksort with *cutoff* for small arrays, as follows: when the size of the array to be sorted falls below $M$, switch to the Insertionsort algorithm. The value of $M$ can be chosen between 10 and 20. (This cutoff trick speeds up the implementation because Insertionsort is faster than Quicksort when the array is small.)

2. The following function promises to rearrange the array `v[p..r]` in increasing order. Show that the function is correct. Estimate the time consumption of the function.

```
void psort (int v[], int p, int r) {
   if (p >= r) return;
   if (v[p] > v[r]) {
      int t = v[p], v[p] = v[r], v[r] = t; }
   psort (v, p, r-1);
   psort (v, p+1, r); }
```

3. PERFORMANCE TEST. Write a program to time your implementation of the Quicksort algorithm. (See the analogous exercise for Mergesort.)

## Height of the execution stack of Quicksort

In the basic version of Quicksort, the code takes care immediately of the segment `v[p..j-1]` of the array and deals with the segment `v[j+1..r]` only after `v[p..j-1]` has been sorted. Depending on the value of `j` in the successive calls to the function, the execution stack can increase a lot, reaching a height equal to the size of the array. (This happens, for example, if the array is originally in decreasing order.) This phenomenon can exhaust the internal memory. To avoid the excessive growth of the execution stack, we must make two patches:

1. deal first with the ⚠ *smaller* of the segments `v[p..j-1]` and `v[j+1..r]` and
2. eliminate the second recursive call of the function, by replacing the "`if`" with a "`while`".

After these two patches, the size of the array segment at the top of the execution stack will be smaller than one half of the size of the segment which is immediately below the top in the stack. More generally, the segment which is at any position of the execution stack will be smaller than one half of the segment immediately below. In this way, when the function processes an array with $n$ elements, the height of the stack will stay below $\log n$.

```
// This function rearranges the array
// v[p..r] in increasing order.

void
quickSort (int v[], int p, int r)
{
   while (p < r) {
      int j = partition (v, p, r);
      if (j - p < r - j) {
         quickSort (v, p, j-1);
         p = j + 1;
      } else {
         quickSort (v, j+1, r);
         r = j - 1;
      }
   }
}
```

## Exercises 5

1. Suppose that the following function is applied to an array with $n$ elements. Show that the height of the execution stack can grow beyond $\log n$.

```
void quicks (int v[], int p, int r) {
   if (p < r) {
      int j = partition (v, p, r);
      if (j - p < r - j) {
         quicks (v, p, j-1);
         quicks (v, j+1, r); }
      else {
         quicks (v, j+1, r);
         quicks (v, p, j-1); } } }
```

2. Study the documentation of the function `qsort` in the `stdlib` library.

See the *Quicksort* entry in Wikipedia.

---

See chapter 11 of the *Programming Pearls* by Jon Bentley.

---

Updated 2019-03-01
https://www.ime.usp.br/~pf/algorithms/
© *Paulo Feofiloff*
*CS-IME-USP*