Sorting Algorithms

One important class of algorithms takes a tuple of mutually comparable elements and puts it in ascending or descending order. These are called sorting algorithms.

Mathematically, we are given a tuple $(a_1, \ldots, a_k)$ of elements of a *totally ordered* set A and we are required to find the permutation σ in the symmetric group acting on numbers from 1 to k such that

$$a_{\sigma(1)} \leq a_{\sigma(2)} \leq \cdots a_{\sigma(k)}$$

where $\leq$ denotes the order.

What is an order on a set?

Recall: A binary relation $\leq$ on a set A is called a *total order* if:

- Given a and b in A either $a \leq b$ or $b \leq a$. (This is called *totality*.)
- The condition $a \leq a$ holds. (This is called *reflexivity*.)
- If $a \leq b$ and $b \leq a$ both hold then $a = b$. (This is called *symmetry*.)
- If $a \leq b$ and $b \leq c$, then $a \leq c$. (This is called *transitivity*.)

Sometimes, we talk of a *strict* order $<$ on the set A. Such a relation $<$ is provided by the relation $(a \leq b)\text{AND}(a \neq b)$. It satisfies the *anti-symmetry* condition that $a < b$ and $b < a$ *cannot* both hold. (Which implies that $a < a$ cannot hold either!) Conversely, given such a strict order $<$, we can define the binary relation $(a < b)\text{OR}(a = b)$ to be the binary relation $\leq$ to get an order as defined above.

Further note that we are using the $a \leq b$ as the notation for the order, but this does not need to be interpreted as "a is less than or equal to b". It can equally well be interpreted as "a is greater than or equal to b" and there is no difference in the above axioms!

There are many algorithms that have been proposed for sorting, so we shall only discuss *some* of them.

Our mathematical description of sorting already leads us to avoid one common programming mistake which is to copy the data many times (e.g. the elements of A may be large objects so copying can take time). This is by limiting ourselves to finding the permutation. That alone is usually enough!

Another aspect that the mathematical description makes clear is that the complexity of the algorithm will *primarily* be obtained by counting the number of *comparisons* that we make. In other words, we will assume that there is a basic operation `cmp` that takes as input two elements a and b of A and returns `True` if $a \leq b$ and `False` otherwise. We will compute the complexity of an algorithm by counting the number of invocations of `cmp`.

Note that when the set A consists of "large" objects, the `cmp` operation can be time-consuming. For example, the order between multi-digit numbers is based

on the *dictionary order* between tuples $(a_1, \cdots, a_n)$ and $(b_1, \ldots, b_n)$ of digits defined as follows. The relation $(a_1, \ldots, a_n) < (b_1, \ldots, b_n)$ holds if and only if

$$a_k < b_k \text{ for some } k, \text{ and } a_i = b_i \text{ for } i = 1, \ldots, k-1$$

Exercise: This is the strict version of the dictionary order. Work out the non-strict version in terms of the elements of the tuple.

Such a dictionary comparison could involve (in the worst case) n comparisons of digits to get the comparison between two tuples.

However, we will *only* count each invocation of `cmp` as *one* unit of time to get the time complexity of our sorting algorithm. We want to know how the number of such invocations depends on the number k of elements of A that we are given.

Bubble Sort

This algorithm is based on the simplest proof that any permutation is a product of transpositions (also called binary swaps).

Each "pass" of the algorithm reads the elements of the tuple one-by-one and swaps as required; a count of the number of swaps performed is kept. In extra pass is performed until there is a pass with no swaps.

define `bubblepass`($\underline{a}$)
  $c = 0$
  set k *to be the length of* $\underline{a}$
  for i *in the range* $[1, k-1]$
    if `cmp`(a_i, a_{i+1}) *returns* `False`.
      increment c *by* 1
      swap a_i *and* a_{i+1}
  return $c, \underline{a}$

define `bubblesort`($\underline{a}$)
  $c = 1$
  while $c > 0$
    $c, \underline{a} =$ `bubblepass`($\underline{a}$)
  return $\underline{a}$

Note that in `bubblepass`, if $k = 1$, then there are no i's in the range $[1, k-1]$. So the "for loop" will not be run at all!

Analysis. We note that `bubblesort` returns $\underline{a}$ when the call to `bubblepass` returns the count c as 0.

So we need to understand when `bubblepass` does that.

Define $|\underline{a}|$ to be the largest i with $a_i \not\leq a_{i+1}$ if there is at least one such i and 0 otherwise.

The first point to note is that $|\underline{a}| = 0$ means that $a_i \leq a_{i+1}$ for *all* i. In this case $\underline{a}$ is ordered.

If $|\underline{a}| = p > 0$, then p is the largest index i in $[1, k-1]$ such that $a_i > a_{i+1}$. In that case $(a_{p+1}, \ldots, a_k)$ are already in the correct order. Any swaps for indices i less than p will only replace a_p by a (possibly) larger element of A. So at index $i = p$ a swap *will* take place. This will mean that $c > 0$. Moreover, after the swap we will compare the old a_p which is now a_{p+1} with a_{p+2} and swap if it is large, and so on. Each time it will "bubble" up if it is larger. Thus, at the end of this call to `bubblepass` the old a_p will "find" its correct place inside the old tuple $(a_{p+1}, \ldots, a_k)$. In other words, $(a_p, a_{p+1}, \ldots, a_k)$ will be sorted at the end of this pass.

Thus, $|\underline{a}|$ is decreasing after each call to `bubblepass` as long as it is greater than 0. So the algorithm `bubblesort` terminates in at most $|\underline{a}| + 1$ calls to `bubblepass`. (The extra 1 is for the call which returns $c = 0$ and makes no change!)

Moreover, each call to `bubblepass` makes $k - 1$ calls to `cmp`.

In the worst case, $|\underline{a}|$ starts with the value $k - 1$ and decrements *exactly* by 1 during each call to `bubblepass`. So the complexity of the algorithm is $k(k-1)$ in terms of calls to `cmp`.

Insert Sort

The above analysis suggests a different approach to sorting as follows.

Given the tuple $\underline{a}$, we note that the analysis above remains correct if *no* swaps were done for indices i such that $i < p$!

This suggests that try to build a bigger and bigger sorted tuple *at one end* by inserting each new element into this tuple. This uses a helper function `insert` that inserts an element b into a *sorted* tuple $\underline{a}$ by starting at the top of the tuple. (We could have equally started at the bottom of the tuple, but we want to re-use the above analysis!)

define `insert`$(\underline{a}, b)$
  set k *to be the length of* $\underline{a}$
  extend the tuple $\underline{a}$ *by adding* b *at the end*
  set j *to be* k.
  while $j \geq 1$ *and* `cmp`(a_j, a_{j+1}) *returns* `False`
    swap a_j *and* a_{j+1}
    decrement j
  return $\underline{a}$

define `insertsort`$(\underline{a})$
  set k *to be the length of* $\underline{a}$
  for i *in the range* $[2, k]$

write $\underline{a}$ *as* $\underline{a'}$ *followed by* a_i *followed by* $\underline{a''}$.
set $\underline{a'}$ *as the output of* `insert`$(\underline{a'}, a_i)$
combine $\underline{a'}$ *with* $\underline{a''}$ *to get the new* $\underline{a}$.
return $\underline{a}$

Analysis. We follow the ideas of the previous analysis with a slight modification. Given a tuple $\underline{a}$ let $v(\underline{a}) = i$ be the smallest index i such that $(a_1, \ldots, a_i)$ is *not* sorted; we can put $v(\underline{a}) = k + 1$ if the full tuple $(a_1, \ldots, a_k)$ is already sorted. Now, $(a_1, \ldots, a_{i-1})$ *is* sorted and `insert` is a procedure that *inserts* a_i into this sorted tuple. The new $\underline{a}$ that is obtained *after* this has a value at least 1 more.

Note that a tuple of length 1 is always sorted, so $v(\underline{a}) \geq 2$, which explains why the loop in `insertsort` starts with 2.

The tuple $\underline{a'}$ input to `insert` has length $k = v(\underline{a}) - 1$, and there are k calls to `cmp` in `insert`. In the worst case $v(\underline{a})$ starts with the value 2 and increases by 1 each time to reach $k + 1$. Thus, the number of calls to `cmp` is the sum of the numbers from 1 to $k - 1$. In other words, the complexity is $k(k-1)/2$.

Further improvements

First of all, we note that the original mathematical statement asked for an algorithm to produce a permutation σ such that $a_{\sigma(1)}, a_{\sigma(2)}, \ldots, a_{\sigma(k)}$ is sorted. Let us first modify the above pseudo-code to produce such a permutation.

We first define `insert`. It takes as input $\underline{a}$, a permutation σ and an index i with the property that the tuple $(a_1, \ldots, a_{i-1})$ is sorted by the permutation $(\sigma(1), \ldots, \sigma(i-1))$. We want `insert` to return the permutation τ such that $(\tau(1), \ldots, \tau(i))$ sorts the tuple $(a_1, \ldots, a_i)$. We will assume that the permutation σ is identity on $(i, \ldots, k)$.

The idea is that we "move up" all the indices j where $a_{\sigma(j)} \not\leq a_i$). This "creates space" to insert the index i just above the first index j for which $a_{\sigma(j)} \leq a_i$.

define `insert`$(\underline{a}, \sigma, i)$
  Set τ *to be* σ.
  Set $j = i - 1$.
  while $(j \geq 1)$ *and* `cmp`$(a_{\sigma(j)}, a_i)$ *is* `False`.
    Set $\tau(j+1) = \sigma(j)$.
    Decrement j.
  Set $\tau(j+1) = i$.
  return τ.

The main routine `insertsort` starts with the identity permutation and repeatedly calls `insert` to increase the applicability to sort the given input.

define `insertsort`$(\underline{a})$
  Set k *to be the length of* (a).
  Set σ *to be the range* $[1, \bar{k}]$.

for *i in the range* $[2, k]$.
  Set $\sigma =$ `insert`$(\underline{a}, \sigma, i)$.
return σ

A similar modification can be made to `bubblesort` and `bubblepass` so that the output is a permutation. We leave this as an exercise. Another approach is to start with the usual tuple of integers $[1, k]$ and pretend that $\underline{a}$ provides a new way to compare integers! So in the algorithm we replace `cmp` by a comparator function `cmpalt` that is defined as follows.

define `cmpalt`$(\underline{a}, i, j)$
  return `cmp`(a_i, a_j)

After this we will only be swapping integers using this different comparator function that depends in $\underline{a}$.

Even more improvements

Note that our algorithms so far "generate" the permutation as a succession of transpositions of the form $(j, j+1)$. However, if we directly view the result of `insert` as multiplication by a single permutation, it is a *cycle* that shifts i to its correct place by shifting others up!

This leads us to wonder if it is possible to *directly* find the position of each element one-by-one. The idea is as follows. Given an element a_p in the tuple $(a_1, \ldots, a_k)$ suppose we find σ so that $(a_{\sigma(1)}, \ldots, a_{\sigma(i-1)})$ are the elements a such that $a \leq a_p$ and $(a_{\sigma(i+1)}, \ldots, a_{\sigma(k)})$ are the elements a such that $a \not\leq a_p$. In that case, $\sigma(i) = p$ *and* the correct place for a_p is at the i-th place. We have now divided the tuple into two shorter tuples; sorting these *independently* and combining them with a_p in the middle will give the required solution.

The above brief description can be converted into the `quicksort` algorithm by using a helper function `partition` that divides the tuple into two tuples. Note that this is a *divide-and-conquer* approach. In the *best* case, the smaller tuples have length roughly $k/2$ so after $\log_2(k)$ steps we will reach tuples of length 1 which are already sorted.

This also suggests a different *divide-and-conquer* approach. We divide the tuple into two (roughly) equal parts, sort those independently and then *merge* two sorted tuples. This approach is called the `mergesort` algorithm and it uses a helper function `merge` that merges two sorted tuples. The idea behind `merge` is to generalise `insert`. The latter can be thought of as `merge` when one of the tuples if of length 1!

Partitioning

Given a tuple $(a_1, \ldots, a_k)$ and an index p, we want an algorithm `parition` that produces a permutation of the tuple

$$(a_{\sigma(1)}, \ldots, a_{\sigma(i-1)}, a_{\sigma(i)=p}, a_{\sigma(i_1)}, \ldots, a_{\sigma(k)})$$

so that for $j < i$, we have $a_{\sigma(j)} \leq a_p$, and for $j > i$ we have $a_{\sigma(j)} \not\leq a_p$.

We want to do this with as few calls to `cmp` as possible.

define `partition`($\underline{a}, p$)
  Set k to the length of $\underline{a}$.
  Set ρ to be the empty tuple.
  Set λ to be the empty tuple.
  Set c to be the element a_p.
  for *i in* $[1, k]$
    Skip to next i if i is p.
    if `cmp`(a_i, c)
      Append a_i to λ.
    else
      Append a_i to ρ.
  return λ, ρ

We note that each element of the tuple is compared with a_p exactly once! This means that there are $k - 1$ comparisons, where k is the size of the tuple $\underline{a}$.

Quicksort Algorithm

This helper function can now be used to perform `quicksort` using the following algorithm. We choose the index p as 1 (which is, on the face of it, no worse than any other choice!).

define `quicksort`($\underline{a}$)
  Set k to the length of $\underline{a}$.
  if $k \leq 1$
    return $\underline{a}$
  Set c to be the element a_1.
  Set $\lambda, \rho =$ `partition`$(\underline{a}, 1)$.
  Set λ to be the output of `quicksort`(λ).
  Set ρ to be the output of `quicksort`(ρ).
  return *λ followed by the singleton c followed by ρ.*

As mentioned above, this works by partitioning the tuple into two parts and sorting each part separately.

When the partitioning is into roughly equal parts at each call to `partition`, there are roughly $\log_2(k)$ "loops". In each loop there are $k - 1$ comparisons. So, in the best case, this algorithm makes roughly $(k - 1)\log_2(k)$ comparisons.

In the worst case (e.g. when the tuples is reverse sorted), we will make $(k - 1)^2$ comparisons!

Some people may suggest that we should take a "random" index p rather than 1 every time. We will discuss such algorithms in a later section.

Merging sorted tuples

We look at a method to merge two sorted tuples that makes at most as many comparisons as the sum of the lengths of the two tuples.

define `merge`$(\underline{a}, \underline{b})$
  Set i *and* j *to be* 1.
  Set p *and* q *to be the lengths of* $\underline{a}$ *and* $\underline{b}$ *respectively.*
  Initialise the answer tuple $\underline{c}$ *as empty.*
  while $i \leq p$ *and* $j \leq q$
    if `cmp`(a_i, b_j)
      Append a_i *to* $\underline{c}$.
      Increment i.
    else
      Append b_j *to* $\underline{c}$.
      Increment j.
  if $j > q$
    return $\underline{c}$ *followed by* $(a_i, \ldots, a_p)$.
  else
    return $\underline{c}$ *followed by* $(b_j, \ldots, b_q)$.

At each point when we enter the bigger **while** loop, a_i is the smallest element of $\underline{a}$ that is not already in $\underline{c}$, and b_j is the smallest element of $\underline{b}$ that is not already in $\underline{c}$. This is the loop invariant property. The algorithm keeps copying the smaller one of these two to $\underline{c}$ until we exhaust either $\underline{a}$ or $\underline{b}$. At that point we only need to append the remaining elements (if any).

In each loop, there is only one comparison and either i is incremented or j is incremented, Since i goes from 1 to p and j goes from 1 to q, there are at most $p + q$ comparisons.

Merge Sort. Using the above helper function, the `mergesort` algorithm is very simple. We divide the tuple into roughly equal halves, sort each smaller tuple and then use `merge` to merge them.

define `mergesort`$(\underline{a})$
  Set k *to be the length of* $\underline{a}$.
  if $k \leq 1$
    return $\underline{a}$
  Set k_2 *to be the integer part of* $k/2$.
  Put $\underline{b}$ *as the first* k_2 *elements of* $\underline{a}$.
  Put $\underline{c}$ *as the remaining* $k - k_2$ *elements of* $\underline{a}$.
  return *the output of* `merge`$($`mergesort`$(\underline{b})$, `mergesort`$(\underline{c}))$.

It is clear that at each "round" the length is being divided by 2 so there are roughly $\log_2(k)$ rounds. Since each round involves a call to `merge` there are at most k comparisons in each round. Thus, the algorithm makes at most $k \log_2(k)$ comparisons.

Lower Bound

Given an algorithm $\mathcal{A}$ that makes N comparisons (and has no other "branching") how may *different* outcomes can be obtained? It is clear that each comparison can lead to at most 2 different sets of outcomes. Thus, the total number of different outcomes of A is at most 2^N.

Given an input of length k to a sorting algorithm, we can get $k!$ (factorial of k) distinct outcomes. For example, the input could be the elements of the set $A = [1, k]$ is *any* permutation order and our sorting algorithm would then produce the inverse permutation which is an element of the permutation group of A.

Thus, if a sorting algorithm that uses *only* comparisons for branching which uses at most N comparisons on every input of size k, we must have the inequality $2^N \geq k!$. Taking $\log_2$ of both sides we see that

$$N \geq \log_2(k!) = \sum_{i=1}^{k} \log_2(i)$$

Now, we have the estimate:

$$\sum_{i=1}^{k} \log_2(i) \leq \sum_{i=1}^{k} \log_2(k) = k \log_2(k)$$

and also the estimate

$$\sum_{i=1}^{k} \log_2(i) \geq \sum_{i=k/2}^{k} \log_2(k/2) = (k/2) \log_2(k/2)$$

From these two we see easily that $\log_2(k!)$ is $\Theta(k \log_2(k))$.

From this we see that the comparison count for `mergesort` is asymptotically growing at the optimal rate! This does not meat that `mergesort` is the optimal algorithm, We can look at improving various constants involved!