Sorting

The sorting problem is defined as follows:

> SORTING: Given a list a with n elements possessing a total order, return a list with the same elements in non-decreasing order.

Remember that *total order* means that any two elements a and b can be compared and we either have $a < b$, or $a = b$, or $a > b$. We only discuss sorting algorithms that work by comparing pairs of elements, and so the same algorithm could be applied to strings, integers, floating point numbers, etc.

Sorting is one of the most fundamental problems in computer science. Here are a few reasons:

- Often the need to sort information is inherent in an application. For instance, you want to report the files that use all the space on your hard disk in decreasing order of size.
- Algorithms often use sorting as a key subroutine. For example, consider the problem of checking whether a list contains duplicated data: The first of the following two algorithms takes $O(n^2)$ time, while the second one uses sorting and then takes only linear time.

```
def has_duplicates(a):
  for i in range(len(a)):
    for j in range(i+1, len(a)):
      if a[i] == a[j]:
        return True
  return False

# This function assumes that a is sorted!
def has_duplicates_sorted(a):
  for i in range(len(a)-1):
    if a[i] == a[i+1]:
      return True
  return False
```

- There are a wide variety of sorting algorithms, and they use a rich set of techniques, as you can see in the following sections. In fact, many important techniques used throughout algorithm design are represented in the body of sorting algorithms that have been developed over the years.
- Sorting is a problem for which we can prove a non-trivial lower bound.

For simplicity, we will explain the algorithms by sorting integer lists, but they all work for arbitrary elements with a total order.

Selection Sort

We already know how to find the minimum of a list of values. And since we are also Masters of Recursion, that means we can sort: We find the smallest element, recursively sort the rest of the list, and concatenate the two pieces:

```
def selection_sort(a):
  if len(a) <= 1:
    return a
  k = find_min_index(a)
  b = selection_sort(a[:k] + a[k+1:])
  return [a[k]]+b
```

Note the base case: a list with zero or one elements is already sorted!

What's the running time of selection sort? `find_min_index` takes $O(n)$ time, so we get the following recursion formula for the running time $T(n)$ of selection sort:

$$T(n) = \begin{cases} O(1) & \text{for } n \leqslant 1 \\ T(n-1) + O(n) & \text{else} \end{cases}$$

The solution is $T(n) = O(n^2)$.

In-place sorting

Our definition of sorting above requires us to return the elements in a new, sorted list. Often this is not necessary, because we no longer need the original, unsorted data. When sorting a huge data set, we can save a lot of memory space by sorting the data *in-place*. This means that the sorting is performed without extra storage (or only little of it). Of course, this is only possible if the data is in a mutable data structure. We will use Python lists:

> InPlaceSorting: Given a Python list a with n elements possessing a total order, rearrange the elements inside the list into non-decreasing order.

Selection sort can easily be rewritten as an in-place sorting algorithm:

```
def selection_sort(a, i):
  if j - i <= 1:
    return
  k = find_min_index(a, i)
  # exchange a[i] and a[k]
  t = a[i]
  a[i] = a[k]
  a[k] = t
  # sort the rest
  selection_sort(a, i+1)
```

The running time is still $O(n^2)$, of course. In fact, this implementation hardly counts as in-place, since it needs n stack frames, which take quite a bit of memory. We should therefore switch to an iterative version (this is easy, because we already have tail recursion):

```
def selection_sort(a):
  n = len(a)
  for i in range(0, n-1):
    # elements 0..i-1 are smallest elements in sorted order
    k = find_min_index(a, i)
    # exchange a[i] and a[k]
    t = a[i]
    a[i] = a[k]
    a[k] = t
```

To argue that this algorithm is correct, we use the loop invariant that elements at index `0` to `i-1` are the smallest elements of the list and already in sorted order.

Insertion Sort

Insertion sort uses recursion the other way round: we recursively sort $n-1$ elements, and finally insert the remaining element into the sorted list. It is based on the observation that it is easy to insert a new element into a sorted list. Here is a version that keeps the original data intact:

```
def sorted_linear_search(a, x):
  for i in range(len(a)):
    if a[i] >= x:
      return i
  return len(a)

def insertion_sort(a):
  if len(a) <= 1:
    return a
  b = insertion_sort(a[:-1])
  k = sorted_linear_search(b, a[-1])
  b.insert(k, a[-1])
  return b
```

The running time of `sorted_linear_search` is clearly $O(n)$, and therefore the running time of `insertion_sort` is again $O(n^2)$.

Let's make insertion sort in-place:

```
# sort a[:j]
def insertion_sort(a, j):
  if j <= 1:
    return
  insertion_sort(a, j-1)
  k = j-1     # remaining element index
  x = a[k]    # value of remaining element
  while k > 0 and a[k-1] > x:
    a[k] = a[k-1]
    k -= 1
  a[k] = x
```

Note how we search for the correct position to place `x` and move the other elements to the side at the same time.

This version is still recursive and will therefore cause a runtime stack overflow quickly. Even though it does not use tail recursion, it's still quite easy to switch to iteration instead of recursion:

```
def insertion_sort(a):
  for j in range(2, len(a)+1):
    # a[:j-1] is already sorted
    k = j-1        # remaining element index
    x = a[k]       # value of remaining element
    while k > 0 and a[k-1] > x:
      a[k] = a[k-1]
      k -= 1
    a[k] = x
```

Note the loop invariant that allows us to argue correctness of this function: At the beginning of the loop, the first $j-1$ elements of the list are already sorted.

Bubble Sort

A very simple in-place sorting algorithm is bubble sort. It's called "bubble sort" because large elements "rise" to the end of the array like bubbles in a carbonated drink.

What makes it so simple is the fact that it only uses exchanges of adjacent elements:

```
def bubble_sort(a):
  for last in range(len(a), 1, -1):
    # bubble max in a[:last] to a[last-1]
    for j in range(last-1):
      if a[j] > a[j+1]:
        t = a[j]
        a[j] = a[j+1]
        a[j+1] = t
```

To prove correctness, we can use the loop invariant that at the beginning of the outer loop the elements at index `last` to `len(a)-1` are already the largest elements of the list in sorted order.

The running time is clearly quadratic.

Bubble sort with early termination One observation about bubble sort is that we can stop once a bubble phase has made no more change—then we know that the array is already in sorted order.

```
def bubble_sort(a):
  for last in range(len(a), 1, -1):
    # bubble max in a[:last] to a[last-1]
    flipped = False
    for j in range(last-1):
      if a[j] > a[j+1]:
        flipped = True
        t = a[j]
        a[j] = a[j+1]
        a[j+1] = t
    if not flipped:
      return
```

Does this improve the time complexity of the algorithm? In the best case, when the input data is already sorted, the running time improves from $O(n^2)$ to $O(n)$. The case of sorted or nearly-sorted input is important, so this is an important improvement.

Unfortunately, in the worst case early termination does not help. The reason is that in every bubble round, the smallest element in the list can only move one position down. So if we start with any list where the smallest element is in the last position, it must take $n-1$ bubble rounds to finish. And therefore bubble sort with early termination still takes quadratic time in the worst case.

Merge-Sort

All the sorting algorithms we have seen so far have a time complexity of $O(n^2)$. To beat this quadratic bound, we need to go back to the idea of *divide and conquer* that we used for the *Maximum Contiguous Subsequence Sum* problem and for binary search.

Recall that divide-and-conquer consists of the following three steps:

- Split the problem into smaller instances.
- Recursively solve the subproblems.
- Combine the solutions to solve the original problem.

Merge-Sort is a sorting algorithms that uses divide-and-conquer as follows: We split the list into two halves, sort each sublist recursively, and then merge the two sorted lists.

```
def merge_sort(a):
  if len(a) <= 1:
    return a
  mid = len(a) // 2
  return merge(merge_sort(a[:mid]), merge_sort(a[mid:]))
```

How do we merge two lists b and b into a new list s? The first element of s must be either the first element of a, or the first element of b. So we compare these two elements, choose the smaller one for s, and continue:

```
def merge(a, b):
  i = 0
  j = 0
  res = []
  while i < len(a) and j < len(b):
    va = a[i]
    vb = b[j]
    if va <= vb:
      res.append(va)
      i += 1
    else:
      res.append(vb)
      j += 1
  # now just copy remaining elements
  # (only one of these can be non-empty)
  res.extend(a[i:])
  res.extend(b[j:])
  return res
```

The running time of `merge` is $O(n)$, where n is the total length of the two input lists. This is true because in every iteration of the `while` loop, the size of `s` increases by one, and in the end `s` has length n.

Let now $T(n)$ denote the number of primitive operations to sort n elements using Merge-Sort. We have the following recurrence relation:

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(\frac{n}{2}) + cn & \text{otherwise.} \end{cases}$$

The recursion looks familiar from the *Maximum Contiguous Subsequence Sum* analysis. The solution is $O(n \log n)$.

Analysis using a recursion tree. We can also analyze the time complexity of Merge-Sort using a *recursion tree*. A recursion tree represents the (recursive) function calls performed during the execution of some program. Each node represents the execution of a function. If the function makes recursive calls, the execution of those recursive calls become children of the node.

The recursion tree for Merge-Sort looks as in Fig. 1. At the root, we work on a list of length n. This is split into two lists of length $n/2$, which are handled at the two children of the node. Each of them makes two calls for lists of length $n/4$, which become their children, and so on.

We have marked in red the running time of each function execution, but *not counting the recursive calls.* At the root, we work on a list of length n and so spend time cn. The children of the root work on a list of length $n/2$, and so the time spent is $cn/2$. Going down each level, the time spent inside a function execution decreases to half. At the bottom level, we handle lists of length one. No recursive call is necessary, and the function returns after running for time c.

Since the subproblem size decreases by a factor two from one level of the tree to the next, the height of the tree is $\log n$. On each level of the tree, the total size of the input lists for all the subproblems on this level is n, and so the total running time of all the functions on this level is cn. It follows that the total running time of Merge-Sort on a list of length n is bounded by $cn \log n$.

We thus have a second proof that the time complexity of Merge-Sort is $O(n \log n)$.

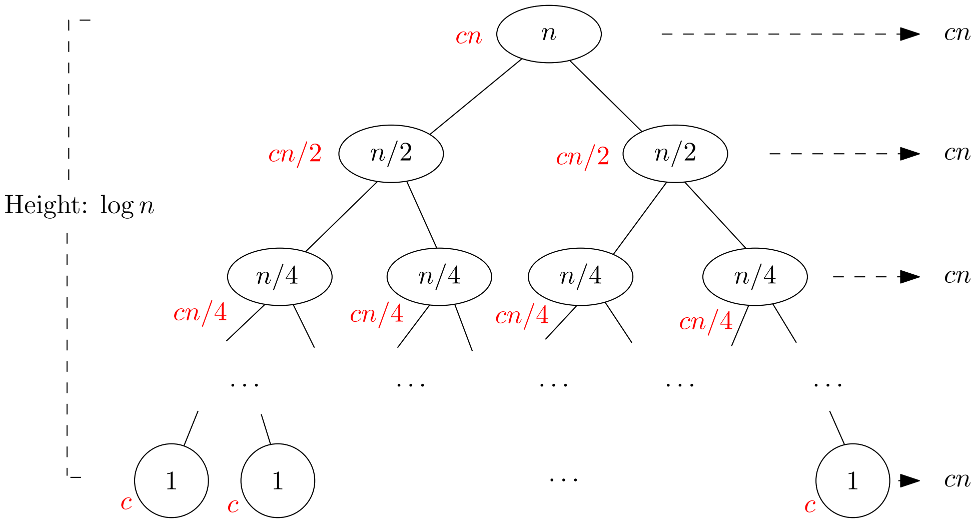

Figure 1: The recursion tree of Merge-Sort.

Quick-Sort

In Merge-Sort, the divide step is trivial, and the combine step is where all the work is done. In Quick-Sort it is the other way round: the combine step is trivial, and all the work is done in the divide step:

1. If L has less than two elements, return. Otherwise, select a *pivot* p from L. Split L into three lists S, E, and G, where
    - S stores the elements of L smaller than x,
    - E stores the elements of L equal to x, and
    - G stores the elements of L greater than x.
2. Recursively sort S and G.
3. Form result by concatenating S, E, and G in this order.

Here is an implementation in Python:

```
def quick_sort(a):
  if len(a) <= 1:
    return a
  pivot = a[len(a) // 2]
  small = []
  equal = []
  large = []
  for x in a:
    if x < pivot:
      small.append(x)
    elif x == pivot:
      equal.append(x)
    else:
      large.append(x)
  return quick_sort(small) + equal + quick_sort(large)
```

Analysis. In the worst case, the running time of Quick-Sort is $O(n^2)$. This happens whenever the partitioning routine produces one subproblem with $n-1$ elements and one with 1 element In other words, each time when we choose the smallest element or the largest element as a pivot, $T(n) = O(1) + T(n-1)$, which gives us the same time complexity as selection sort or insertion sort: $O(n^2)$.

The best case happens when the pivot splits the list into two equal parts S and G. In that case the recurrence is $T(n) = 2T(n/2) + O(n)$, which solves to $O(n \log n)$ as in Merge-Sort.

In old textbooks, the pivot is often chosen as the first element in the list. This is a really bad choice, since in practice lists are often already sorted somewhat. Chosing the first element would then often give us the smallest element in the list.

One good strategy is to chose a pivot randomly (generate a random index into the list and pick that element), as we have done in the code above. With probability 1/2, we will pick a pivot such that neither S nor G has more than $3n/4$ elements. Imagine that this happens in every step: Then the depth of the recursion is still $O(\log n)$, and the recursion tree argument implies that the total running time is $O(n \log n)$. Of course we cannot expect this good case to happen all the time, but on average you should get a good split every second time, and this is enough to argue that the depth of the recursion tree will be $O(\log n)$ with high probability.

The journal *Computing in Science & Engineering* did a poll of experts to make a list of the ten most important and influential algorithms of the twentieth century, and it published a separate article on each of the ten algorithms. Quicksort was one of the ten, and it was surely the simplest algorithm on the list. Quicksort's inventor, Sir C. A. R. "Tony" Hoare, received the ACM Turing Award in 1980 for his work on programming languages, and was conferred the title of Knight Bachelor in March 2000 by Queen Elizabeth II for his contributions to "Computing Science."

In-place Quick-Sort

One advantage of Quick-Sort compared to Merge-Sort is that it can be implemented as an in-place algorithm, needing no extra space except the array storing the elements:

```
# sort range a[lo:hi+1]
def quick_sort(a, lo, hi):
  if (lo < hi):
    pivotIndex = partition(a, lo, hi)
    quick_sort(a, lo, pivotIndex - 1)
    quick_sort(a, pivotIndex + 1, hi)
```

The critical method is `partition`, which choses the pivot and partitions the list into two pieces. This is quite tricky, as we (a) want to do it in-place, (b) have to nicely handle the case when there are many equal elements. It's easy to write a buggy or quadratic version by mistake. Early editions of the Goodrich and Tamassia book did.

We have an array `a` in which we want to sort the items starting at `a[lo]` and ending at `a[hi]`. We choose a pivot index `p` and move it out of the way by swapping it with the last item, `a[hi]`.

We employ two array indices, `i` and `j`. `i` is initially `lo - 1`, and `j` is initially `hi`, so that `i` and `j` sandwich the items to be sorted (not including the pivot). We will enforce the following loop invariants (v is the value of the pivot):

- $a[k] \leqslant v$ for $k \leqslant i$,
- $a[k] \geqslant v$ for $k \geqslant j$.

To partition the array, we advance the index `i` until it encounters an item whose key is greater than or equal to the pivot's key; then we decrement the index `j` until it encounters an item whose key is less than or equal to the pivot's key. Then, we swap the items at `i` and `j`. We repeat this sequence until the indices `i` and `j` meet in the middle. Then, we move the pivot back into the middle (by swapping it with the item at index `i`).

What about items having the same key as the pivot? Handling these is particularly tricky. We'd like to put them on a separate list (as in the list version above), but doing that in-place is too complicated. If we put all these items into the first list, we'll have quadratic running time when all the keys in the array are equal, so we don't want to do that either.

The solution is to make sure that each index, `i` and `j`, stops whenever it reaches a key equal to the pivot. Every key equal to the pivot (except perhaps one, if we end with `i = j`) takes part in one swap. Swapping an item equal to the pivot may seem unnecessary, but it has an excellent side effect: if all the items in the array have the same key, half of these items will go into the left part, and half into the right part, giving us a well-balanced recursion tree. (To see why, try running the function below on paper with an array of equal keys.)

```
# partition range a[lo:hi+1] and return index of pivot
def partition(a, lo, hi):
  p = (lo + hi)//2
  pivot = a[p]
  a[p] = a[hi]  # Swap pivot with last item
  a[hi] = pivot

  i = lo - 1
  j = hi
  while i < j:
    i += 1
    while a[i] < pivot:
      i += 1
    j -= 1
    while a[j] > pivot and j > lo:
      j -= 1
    if i < j:
      t = a[i]; a[i] = a[j]; a[j] = t  # swap a[i] and a[j]
  a[hi] = a[i]
  a[i] = pivot # Put pivot where it belongs
  return i     # index of pivot
```

Can the `while a[i] < pivot` loop walk off the end of the list and generate an exception? No, because `a[hi]` contains the pivot, so `i` will stop advancing when `i == hi` (if not sooner). There is no such assurance for `j`, though, so the `while a[j] > pivot` loop explicitly tests whether `j > lo` before decreasing `j`.