Big-O Notation

It is useful to describe the run-time of algorithms as a function of the size of the input, say n. We call this the computational complexity of an algorithm. We care about the behavior of this function as n gets large, and ignore constant factors. In practice, the constant factors may make a big difference. For example, the function 1.01^n grows faster than $10^{10} \cdot n$, but is much smaller at the beginning (and we may never need to consider large values of n in our application).

Big-O notation[1] is often used to denote the order of the run-time. We say $f(n) = O(g(n))$ if there exist positive real numbers C and n_0 such that $g(n) \geq C \cdot f(n)$ for all $n \geq n_0$. In other words, at some point $g(n)$ overtakes $C \cdot f(n)$ forever.

If there is an n_0 for any constant C such that $g(n) \geq C \cdot f(n)$ for all $n \geq n_0$, then we say that $f(n) = o(g(n))$. This means that $g(n)$ grows faster than $f(n)$, so any constant C can only give $f(n)$ a "head start". This is another way of saying that $\frac{f(n)}{g(n)} \to 0$ as $n \to \infty$.

Sorting Algorithms

We discuss and implement 4 different sorting algorithms, so that you can get a feel for estimating the complexity of an algorithm. Most of you will not be theoretical computer scientists, but when you are programming this should always be in the back of your mind, and it is good to have a feel for what a fast or a slow function looks like. We will build the sorting functions "from scratch" as much as we can, so that we can see the inner workings and understand the number of steps each line takes for the computer to execute.

Our sorting functions take as input a list of real numbers, and they will output a list containing the same elements, but in increasing order. (Note that the lists may have duplicates.)

Selection Sort

This is perhaps the first sorting algorithm you would think of, if you have 5 exam papers in front of you that need to be alphabetized. You look them over and pick the one that is first in the alphabet, pick it up, then look for the second, etc. Selection sort moves elements from the original list to the sorted list one by one, always picking a smallest element first. In pseudo-code

```
SELECTION SORT (L)
s=L=empty list
while L:
   find the first smallest element a of L (there may be multiple)
   remove a from L
   add a to the end of sL
return sL
```

Finding the minimum of a list takes $O(n)$ time (if n is the length of the list). We can go through the list element by element and keep track of the smallest one we have seen so far. There is no faster way of doing this, since we must at least look at each element in order to guarantee we find

[1] If you are interested, you can read more here: `http://web.mit.edu/16.070/www/lecture/big_o.pdf`.

the minimum. While we find the minimum, we can immediately keep track of the index of the first one, which then helps us remove it. Overall, each iteration of the loop is $O(n)$. The loop is executed exactly n times (why?), which gives a total run-time of $O(n^2)$. You may notice that the L only has length n at the start, after that it gets shorter and shorter. Does this not give a faster time? On average over all loops, L has length $n/2$, so this only gives us a constant factor improvement.

Bubble Sort

Bubble sort runs through the list and swaps every element with it's right neighbor if the element is greater than the neighbor. It keeps running through the list like this until there are no more available swaps. You can think of this is bubbles bubbling upwards one by one. In a sorted list, we require that $L[i] \leq L[i+1]$ for all $0 \leq i \leq n-2$. However, we should be careful to only swap if there is a strict inequality, rather than a greater than or equal to, since that would lead to an infinite loop. In pseudo-code

```
BUBBLE SORT (L)
sL=L
sorted=False
while not sorted:
   run through sL and swap adjacent unordered elements
   if there are none, set sorted to True
return sL
```

On average, if we assume that our list is uniformly shuffled, about half of the pairs will be out of order. This means that there need to be about $n^2/2$ swaps, giving us an average complexity of $O(n^2)$.

Merge Sort

If instead of five exam papers, you have 200 in front of you, you may realize that selection sort or bubble sort are not the best approaches, and you may come up with something akin to merge sort or quick sort, which split the problem up into smaller problems (this is especially obvious if you have TAs who are helping you sort). This is my algorithm of choice when grading. Merge sort is a recursive algorithm, which splits the list in half, sorts the halves, and then "zips" them back together. In pseudo-code

```
MERGE SORT (L)
if length L >1:
   split L in halves L1 and L2
   sL1= MERGE SORT (L1) and sL2= MERGE SORT (L2)
   zip sL1 and sL2 into sL
   return sL
else:
   return L
```

The "zip" part works as follows. Suppose we have two sorted lists sL1 and sL2. Repeatedly remove the first element from either sL1 or sL2, depending on which one is smaller, and add it to the end of the sorted list. Do this until both sL1 and sL2 are empty. Now, they have been zipped together into a complete, sorted list.

The zipping part of merge sort takes time $O(n)$ for each level (number of times the list has been split), because we always execute one comparison and one move for each element of L. The number of times we split is $\log_2(n)$ (rounded up), since we split until our lists are length 0 or 1. This gives arun-time of $O(n\log(n))$.

Quick Sort

It might make sense to first split the exam papers into a pile with the first half of the alphabet (say, everything up to K), and one with the second pile (everything L and onwards), then have your TAs order those. In that case, the merging part is easy! This is what quick sort does. It is very similar to merge sort, except it adds complexity to the splitting part, which then saves complexity in the merging part.
In pseudo-code

```
QUICK SORT (L)
if length L >1:
   choose a pivot element p from L and remove p from L
   let L1 be remaining elements <p, and L2 remaining elements >=p
   let sL be concatenation of QUICK SORT (L1) and QUICK SORT (L2)
else:
   return L
```

In this case, the splitting part of quick sort takes $O(n)$ time. (Why?) If p is chosen well, then the list is split roughly in half each time. In merge sort, we had control over this, but in quick sort, we do not. Suppose that the list is already sorted. This would be a very good case for bubble sort, but for quick sort, this results in having to split n times instead of $\log n$ times. To avoid this, we often choose p by picking three elements from L uniformly at random, and letting the middle one be p. This gives an expected run-time of $O(n\log n)$[2], even when the list is already sorted.

Exercises

- Write and analyse insertion sort: this algorithm takes the first element of and looks for its rightful place in the sorted lists, and inserts it there.
- Write algorithms that find the minimum, maximum, mean, median and variance from a list of real numbers L. Can you do all of these in $O(n)$?
- Do a complete analysis of the worst and best case scenarios for each algorithm.
- It is important to remove p from L, such that the total length of L1 and L2 is strictly less than L. If not, you may get stuck in an infinite loop. Explain how.

[2]The run-time $O(n\log n)$ is proven to be the best we can do for lists of real numbers in general. However, there are possible improvements (even $O(n)$) when we have information about the set of elements or their range.