Efficiency Issues

An implementation may be correct with respect to the Specification Pre- and Post-condition, but nevertheless be impractical as it consumes too much time, space, etc.

Example: The following informally described algorithm satisfies the Pre- and Post-conditions on sorting an array of integers.

Given the array of integers, systematically generate each possible permutation of it. For each permutation, test to see if it is sorted.

As an ordered list of N items has N! permutations, this algorithm has a run time of O(N!) which is a lot worse than 2**N. But it is *correct*!

Moral: *Correctness Ain't Enough.*

Run Times

It is therefore helpful to have an idea of how long an implementation (algorithm) takes to deliver a result.

To be fair, we will measure this as a function of the length n of its input, and only count high-level operations like assignments, procedure calls, pairwise comparisons, etc. Often it suffices to count comparisons as these dominate the run-time.

The O(_) notation is popular. $O(n^2)$ means that the algorithm takes roughly n^2 steps (e.g. comparisons) to run, where n is the length (e.g. the length of the input array) of the input.

If there is some fixed number k for which the algorithm will run in (at most) $O(n^k)$ steps, it is said to be a *polynomial* (time) algorithm. If it uses $O(2^n)$ (or worse) it is *exponential*.

Run Times cont'd

Algorithms that run in $O(log\ n)$ are *logarithmic*.

Those that run in $O(n)$ are *linear*.

It can sometimes be proved that a certain *Class* of problems do not have linear algorithms, or even polynomial ones.

Worse, there are classes for which it can be shown that there are *no algorithms whatsoever*. An example is the problem of designing a compiler to say "Yes/No" to whether an arbitrary program will stop for an arbitrary input just by examining its syntax.

More on O( ) notation

By $O(n^2)$ is meant the *class* of numeric functions that "grow" asymptotically no faster than n^2, disregarding constant multipliers. So, $n^2 + 25n - 400$ is $O(n^2)$, and so is $37n^2$ and also $3n$.

Likewise for other complexity classes like logarithmic, linear, etc.

A reasonable test to see if f is $O(g)$ is to examine $lim_{n\to\infty} \frac{f(n)}{g(n)}$ to see if it is a constant.

Run Times for Sorting

Consider sorting an array of n integers. We saw BubbleSort and QuickSort. It is not hard to give an informal analysis of their run times.

We will just look at the number of comparisons as the measure.

For BubbleSort, we will look at the *Worst Case* measure. For QuickSort we will do a "hand-waving" *Average Case* measure.

Worst Case means assuming the most unfriendly data that will force the algorithm to work the hardest (make most comparisons). Average Case means "average" or "typical" data.

Run Time for BubbleSort

```
class BubbleSortAlgorithm extends SortAlgorithm {
  void sort(int a[]) {
    for (int i = a.length; --i>=0; )
      for (int j = 0; j<i; j++) {
        if (a[j] > a[j+1])
          swap(a, j, j+1);
        pause();
      }
  }
}
```

For each outer loop index i, the inner j loop does i comparisons. As i ranges from n (=a.length) to 1, the number of comparisons is $n + (n-1) + (n-2) \ldots 3 + 2 + 1 = n(n+1)/2$, which is $O(n^2)$.

QuickSort Skeleton

By way of review:

```
QSort(array of length n)
  Do the Split into two arrays of length n1 and n2
  Recursively call Qsort(array of length n1)
  Recursively call Qsort(array of lenght n2)
End
```


Run Time for QuickSort

Here is the "hand-waving" up front. An "average" array will result in the array being split into approximately two equal halves.

If the run time of Quicksort on an array of length n is denoted by $T(n)$, then we have the recurrence relation:

$$T(n) = T(n/2) + T(n/2) + n$$

where the two $T(n/2)$ terms come from the two recursive calls, and the n is the overhead of *Split*.

From your discrete math course, this recurrence relation has the solution $T(n)$ is $O(n * log\ n)$.

Lower Bound on Sorting

We will now outline an argument that *every* sorting algorithm on n items requires $O(n * log\ n)$ comparisons in the Worst Case.

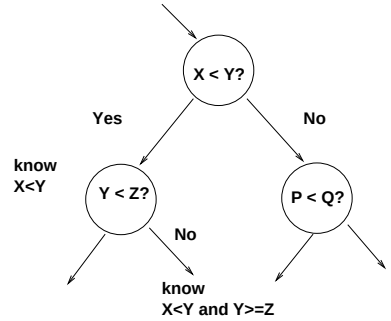
This means that $O(n * log\ n)$ is a *Lower Bound* on sorting. Moreover, any sorting algorithm that achieves this must be *Optimal*.

The style of argument is called "information-theoretic".

We first explain a model of computation.

Tree Model of Computation for Sorting

No matter what language or machine ones uses for sorting, if we are concentrating on *comparisons*. We only have to model comparisons between two numbers (say). This can be done as follows:



Program Executions are Trees

Each possible sequence of comparisons *is* a path in a binary tree.

At each lower node more and more information is known. Each *leaf* of a tree is *complete knowledge* about, say, the ordering relationship between array elements, e.g., $a[7] > a[5] > a[1] \ldots$.

This complete information is *equivalent to sorting the array*.

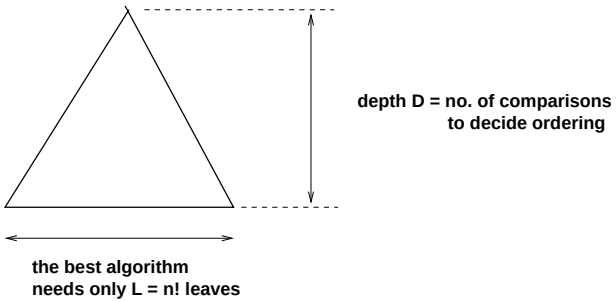
How Many Leaves?

Inputs can be in any order (permutation), and algorithms can compare in any sequence, and also choose to compare any two elements at any stage.

Hence there must be *at least n!* leaves since there are *n!* possible permuations. There could be more, as an algorithm may use two distinct comparison sequences to arrive at the same information.

But the best possible (cleverest, etc!) algorithms may need only $n!$ leaves.

How Deep is a best Tree?



For a binary tree, $L = 2^D$, or $D = log_2 L$. Here $D = log_2 n!$.

Lower Bound of $O(n * log\ n)$

Since this tree is the best any algorithm can do (others will have larger D), $log_2 n!$ is the smallest number of comparisons any algorithm has to use to sort n numbers if the input can be in any order.

Stirling's approximation: for large n:

$$n \simeq (n/e)^n \sqrt{(}2n\pi)$$

Hence the lower bound on sorting is $O(log(n^n)) = O(n * log\ n)$.

Optimal Sorting Algorithms

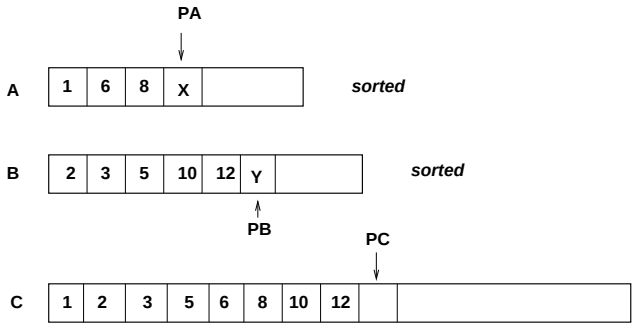
BubbleSort is not optimal. Neither is QuickSort, for it is Worst Case $O(n^2)$ even though Average Case $O(n * log\ n)$.

Are there optimal, therefore $O(n * log\ n)$ sorting algorithms?

Yes: MergeSort and HeapSort (both are also easy to program). Surprisingly, so is QuickSort even though it is not Worst Case optimal.

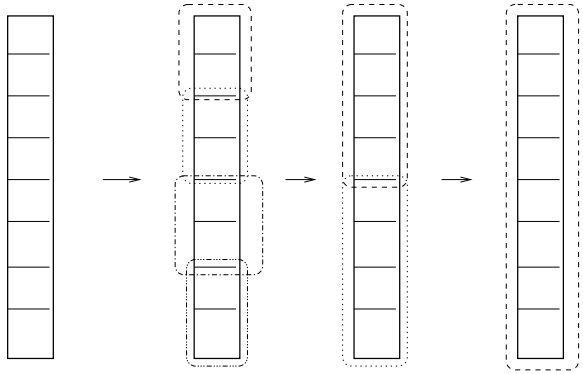
We will describe MergeSort informally.

Merging two sorted arrays



If X<Y then copy X to C[PC]; advance PA
else copy Y to C[PC]; advance PB
Advance PC

MergeSort — Iterative Version



Run Time of MergeSort

How many merges to finish? Within each merge, how many comparisons?

Initially, pair-wise adjacent comparisons to build up 2-length sorted sub-arrays: $n/2$ comparisons (actually, a special case of merging 1-length sub-arrays!). Generally, given two k-length sorted arrays, to merge into one $2k$ array takes $O(2k)$ comparisons.

To get to an n-length sorted array, need $O(log\ n)$ merges, since each merge doubles the length of the sorted array.

So the number of comparisons is $n/2 + n + n + \ldots + n$ with $O(log\ n)$ terms, i.e., $O(n * log\ n)$.

Practical Considerations

The "natural" view of MergeSort is interative, and uses copying. This copying can be expensive in storage and time. A bit of cleverness can avoid the copying.

In practice, Quicksort is the most favored algorithm for internal sorting. It can be tuned to have close to optimal performance by judicious selection of the Pivot.

If an iterative sort is required, HeapSort avoids the copying in Mergesort as well as its requirement for extra memory.

More Rigorous Average Runtime for QuickSort

Let $T(N)$ be the average cost of quicksort on N items. As we do not know the relative sizes of the LEFT and the RIGHT splits as quicksort progresses in its "divide-and-conquer" recursion, let us *average* these costs as:

$$T(LEFT) = T(RIGHT) = \frac{T(0) + T(1) + \ldots + T(N-1)}{N} \quad (1)$$

where the averaging is over the costs of the recursively split pieces.

Thus, when we add the cost of splitting at the top level, and the costs the top level LEFT and RIGHT split pieces:

$$T(N) = 2 * \frac{T(0) + T(1) + \ldots + T(N-1)}{N} + N \quad (2)$$

Cont'd QSort Average Time

Multiplying equation 2 by N yields

$$N * T(N) = 2 * (T(0) + T(1) + \ldots + T(N-1)) + N^2 \quad (3)$$

Similary, for the $N-1$ case we get

$$(N-1) * T(N-1) = 2 * (T(0) + T(1) + \ldots + T(N-2)) + (N-1)^2 \quad (4)$$

Subtracting equation 4 from equation 3, dropping a constant and rearranging, yields:

$$N * T(N) = (N+1) * T(N-1) + 2 * N \quad (5)$$

From this we get, after dividing by $N * (N+1)$:

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2}{N+1} \quad (6)$$

Now *telescope*! (cf. proof of sum of first n numbers without using induction):

$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2}{N+1}$

$\frac{T(N-1)}{N} = \frac{T(N-2)}{N-1} + \frac{2}{N}$

$\frac{T(N-2)}{N-1} = \frac{T(N-3)}{N-2} + \frac{2}{N-1}$

$\ldots$

$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2}{3}$

Adding LHS's and RHS's ("cancelling" across =) gives

$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2 * (\frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{N+1})$

the RHS of which is $O(log\ N)$ (harmonic sum approx by $\int \frac{dx}{x}$).

Therefore $T(N)$ is $O(N * log\ N)$.