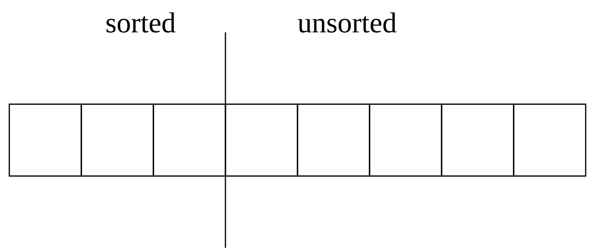
Insertion sort

Insertion sort is perhaps the simplest sorting method. If you ever need to write a sorting function in a hurry, this is the one to choose. It is not a very good method, but it is easy to get right.

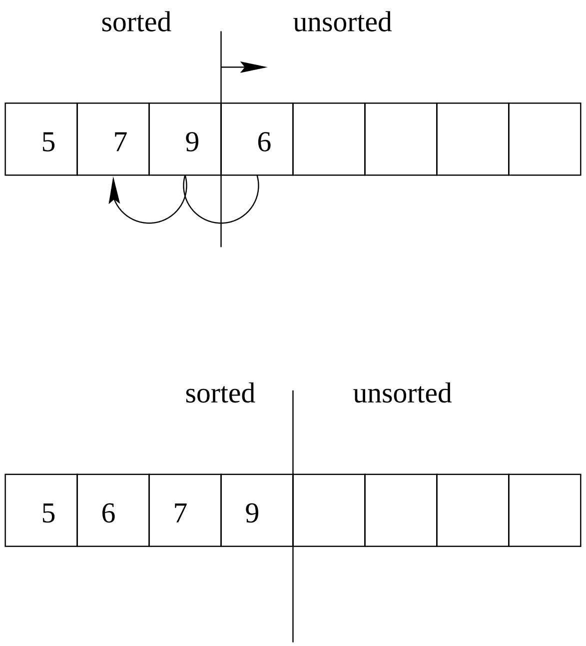
The method works as follows. You are sorting an array. We divide the array into two sections: a sorted section (usually on the left, occupying the small index positions), and an unsorted section (or, rather, a section that we haven't looked at yet). Here's the picture:



One step at a time

Insertion sort starts by placing the sorted-unsorted dividing line between elements 0 and 1 (a single element is always sorted).

The method then advances the line one element to the right, swapping the newly encountered element with its predecessor until it is correctly sorted.



Analysing insertion sort

Insertion sort runs in two loops: an outer loop processes each element in the array, inserting it into position. The inner loop has the task of inserting a single element into position.

The amount of work done by the inner loop depends on how many larger elements already exist in the sorted part of the array. If there are none, the loop completes after a single comparison. If there are many, the loop compares and swaps with each one. In the worst case, every element is larger than the one being inserted (this occurs when the array is initially in reverse-sorted order).

The time for the inner loop, T_{ins}, depends on the size of the sorted array. The time for the outer loop, T_{isort}, (and hence the whole sort) is the sum of these inner loop times.

$$T_{\text{isort}}(n) = T_{\text{ins}}(1) + T_{\text{ins}}(2) + \ldots + T_{\text{ins}}(n-1)$$

If $T_{\text{ins}}(i) = k$ for some constant k (the best case), then

$$T_{\text{isort}}(n) = k(n-1) = \Omega(n)$$

If $T_{\text{ins}}(i) = ki$ (the worst case), then

$$\begin{aligned} T_{\text{isort}}(n) &= \sum_{j=1}^{n-1} kj \\ &= k\frac{n(n-1)}{2} = O(n^2) \end{aligned}$$

On average, $T_{\text{ins}}(i) = \frac{ki}{2}$, which is identical to the worst case (dividing an arbitrary constant by 2 gives another arbitrary constant).

The worst-case (and therefore average case) behaviour of insertion sort is bad indeed, but best-case behaviour is worth remembering.

Analysis of merge sort

We can express its runtime as the recurrence

$$T_{\texttt{mergeSort}}(n) = \begin{cases} \Theta(1) & n \leq 1 \\ 2T_{\texttt{mergeSort}}(n/2) + \Theta(n) & n > 1 \end{cases}$$

It turns out this recurrence has the solution $T_{\texttt{mergeSort}}(n) = \Theta(n \log n)$.
So, merge sort is better than insertion sort in the worst and average cases, but insertion sort has the edge with nearly-sorted data.

Solving more recurrences

We have already observed that a recurrence of the form $T(n) = T(n-1) + \Theta(1)$ gives a linear runtime, and that $T(n) = T(n-1) + \Theta(n)$ gives a quadratic runtime.

Divide-and-conquer algorithms produce recurrences of the form $T(n) = aT(n/b) + f(n)$. I.e., a sub-problems are produced, each of size n/b, and $f(n)$ work is needed to combine the results of the sub-problems.

There is a set of standard solutions to many recurrences of this form, called the "master method". We will look at the special case for merge sort first, then study the more general solution.

Proof of the merge sort recurrence

This proof gives the general flavour of solving divide-and-conquer recurrences. The formal proof is complicated by certain technical details, such as when n is not an integer power of b. We will ignore such issues in this proof, appealing instead to the intuition that runtime will increase in a more-or-less smooth fashion for intermediate values of n.

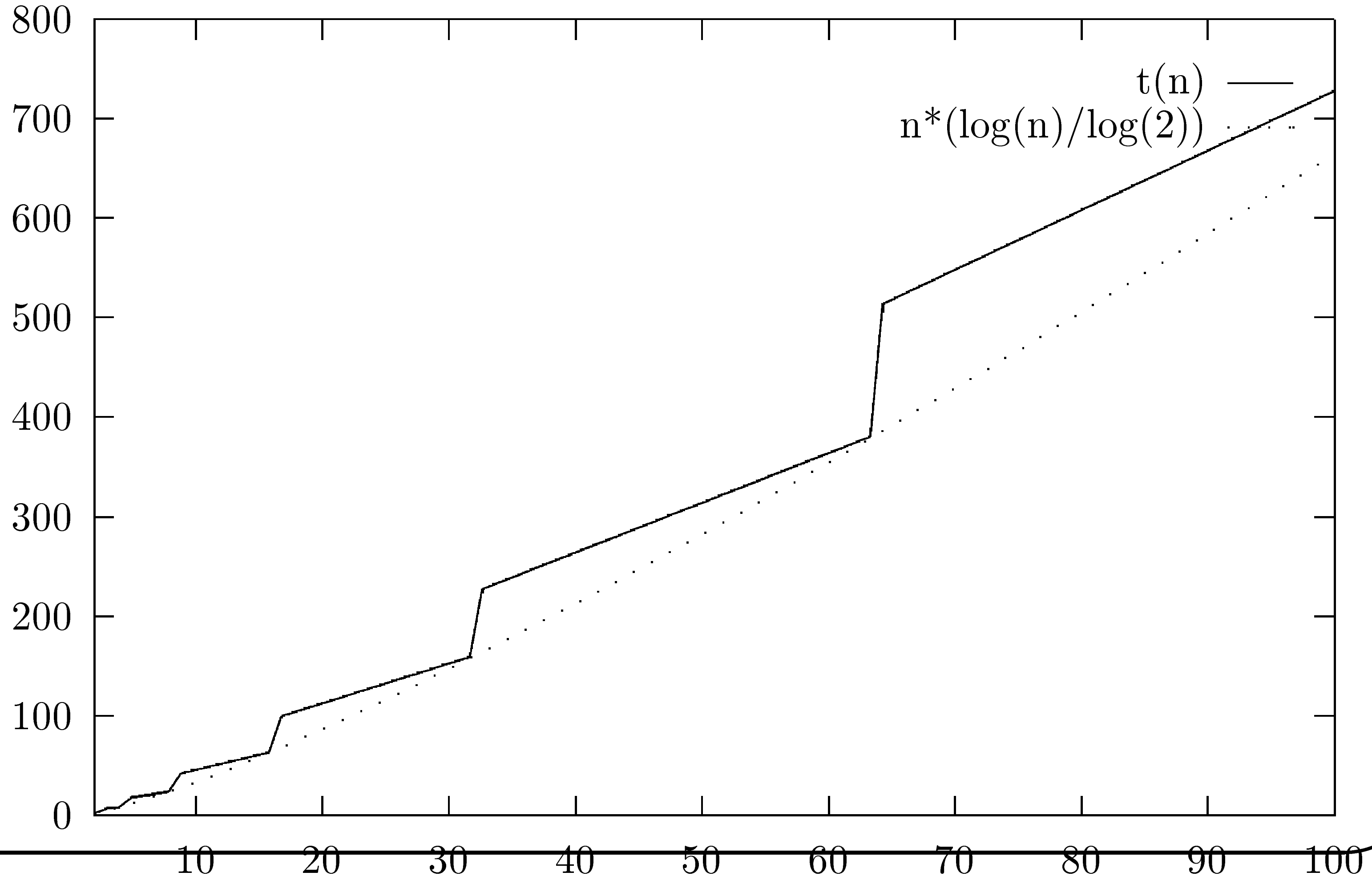
If

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 2T(n/2) + n & \text{if } n = 2^k, k > 1 \end{cases}$$

then $T(n) = n \log n$.

Plotting with gnuplot

```
gnuplot> t(n) = (n<=2) ? 2 : 2*t(n/2)+n
gnuplot> plot [n=2:100] t(n), n*(log(n)/log(2))
```



Proof by induction

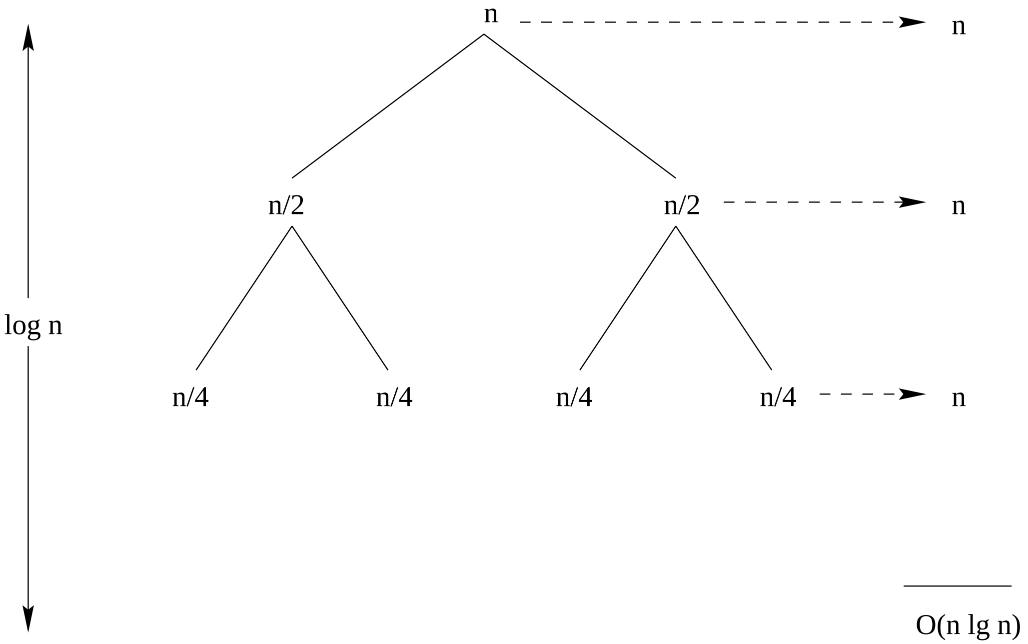
BASIS $T(2) = 2 = 2\log 2$

INDUCTIVE STEP: assume $T(2^k) = 2^k \log 2^k$.

$$
\begin{aligned}
T(2^{k+1}) &= 2T(2^{k+1}/2) + 2^{k+1} \\
&= 2T(2^k) + 2^{k+1} \\
&= 2(2^k \log 2^k) + 2^{k+1} \\
&= 2^{k+1}(\log 2^k + 1) \\
&= 2^{k+1}(\log 2^k + \log 2^1) \\
&= 2^{k+1} \log 2^{k+1}
\end{aligned}
$$

Recursion trees

Recursion trees help to visualise what happens when a recursion is expanded. The recursion tree for the merge sort recurrence looks like this:



The "master theorem"

The solution to a recurrence of the form $T(n) = aT(n/b) + f(n)$ is a battle between the two terms in the sum. Think of $aT(n/b)$ as the process of "distributing the work out" to $f(n)$, where the work is actually performed.

If $f(n)$ is trivial, then $T(n) = \Theta(n^{\log_b a})$.

If $f(n)$ dominates, then $T(n) = \Theta(f(n))$.

If both terms are balanced (which happens when $f(n) = \Theta(n^{log_b a})$), then $T(n) = \Theta(n^{log_b a} \log n)$.

Definition of "dominates"

The three cases of the master theorem depend on comparing $f(n)$ to $n^{\log_b a}$. However, it is not sufficient for $f(n)$ to be "just a bit" bigger or smaller than $n^{\log_b a}$. The master theorem can only apply when there is a *polynomial* difference between the functions.

For example. n^2 is polynomially larger than $n^{1.5}$, but $n \log n$ is *not* polynomially larger than n.

There is also a "regularity" condition on the solution $T(n) = \Theta(f(n))$, that $af(n/b) \leq cf(n)$ for some constant $c < 1$; all the functions we will come across will satisfy this condition.

Gaps in the master theorem

The master theorem does not cover all possible cases. It fails to work when $f(n)$ is close to $\Theta(n^{log_b a})$, but by some factor less than n^ϵ.

For example, the master theorem does not apply to the recurrence $T(n) = 2T(n/2) + n \log n$, despite having the proper form. We have $a = b = 2$, so we need to compare $f(n) = n \log n$ to $n^{\log_b a} = n^1 = n$. Clearly, $f(n)$ is larger than n, but the ratio $f(n)/n$ is $\log n$, which is not larger than any polynomial n^ϵ.

Worst-case analysis of quicksort

A simple version of quick sort chooses the leftmost element as the pivot. If the array is already sorted, the partition step will produce a left sub-array of size 1, and a right sub-array of size $n-1$. The runtime recurrence is thus

$$\begin{aligned} T_{\text{quickSort}}(n) &= T_{\text{quickSort}}(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Best and average case partitioning

If the partition step always produces two equal size regions, the recurrence becomes

$$\begin{aligned} T_{\text{quickSort}}(n) &= 2T_{\text{quickSort}}(n/2) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

If partition produces 9-to-1 splits each time (a poor average behaviour), then the recurrence becomes

$$T_{\text{quickSort}}(n) = T_{\text{quickSort}}(9n/10) + T_{\text{quickSort}}(n/10) + \Theta(n)$$

This split still produces a recursion tree that terminates at depth $\log_{10/9} n = \Theta(\log n)$, so the runtime is still $\Theta(n \log n)$.

What is a "stable" sort?

In practice, the data to be sorted is comprised of two parts: a *key*, which is used for comparison, and some additional *satellite data.* Thus, data has the property that two different elements elements may compare "equal".

For example, you may sort a list of mail messages by the sender, or by the date sent, or by the subject. When sorting by date sent, all messages from a given day will compare equal.

A *stable* sort is one that never exchanges elements with equal keys.

Merge sort and insertion sort are stable; quick sort is not.