Physics 115/242

Sorting Algorithms

Peter Young

(Dated: April 23, 2014)

I. INTRODUCTION

We frequently need to sort a list of numbers. These might, for example, be energy eigenvalues in a quantum mechanics problem, in a situation where the lowest energy level, and the low-lying excited states, determine the behavior of the system and we want to know which they are.

Sorting is also of interest from a computing point of view because it turns out to be an example of a type of problem in which there is a very simple algorithm which is highly inefficient when applied to large data sets, but a much more efficient algorithm can be with some clever thought.

The problem, then, is that we have a set of numbers x_i, $(i = 1, 2, \cdots, N)$ which we want to sort in ascending order. (Note: C programmers, perversely in my view, insist on starting all arrays at 0 even if, as in this example, it is most natural to start at 1. Thus, for a hard-core C programmer one has x_i, $(i = 0, 1, \cdots, N-1)$. The x_i are stored in array which will be fed into our sorting routine. On exit from the routine, the array should contain the numbers in the correct order.

II. SIMPLE BUT INEFFICIENT ALGORITHMS

The simplest method, often called "bubble sort", proceeds as follows. There are two nested loops:

- In the outer loop there is an index i which runs from 1 to $N-1$, (in C from 0 to $N-2$).
- For each value of i, there is an inner loop with an index j which goes from 1 to $N-i$, (in C, j goes from 0 to $N-2-i$). For each value of j, we swap elements x_j and x_{j+1} if $x_j > x_{j+1}$. Thus the inner loop compares data values in the range $1 \leq j \leq N+1-i$, $(0 \leq j \leq N-1-i$ in C), and has the effect of pushing the largest element in this range to the bottom element of the range.

To see why this works, consider the first value of i. Once the loop over j reaches the largest element, subsequent values for j just push this element down until it reaches the bottom, which is its right

place. For the next value of i, the sum over j does not involve the last element which is therefore left in place, and second largest element is run down to the next to the last location, where it will stay for subsequent values of i. Continuing in this way, after i has got to its last value, all the elements are in the correct (ascending) order.

This algorithm is adequate for small N. However, for large N the computer time becomes very large. To see this, note that we have two nested loops each of which runs over N elements (or a number of order N). Hence the number of operations is proportional to N^2. If N is, say, 10^6 this represents a large CPU time, even on modern machines.

There is a somewhat faster, but still very simple, routine, called "straight insertion" in Numerical Recipes (`piksrt`). In my experience it is about 3 times faster than bubble sort. Since the operation count is still of order N^2, it too is unsuitable for sorting large data sets.

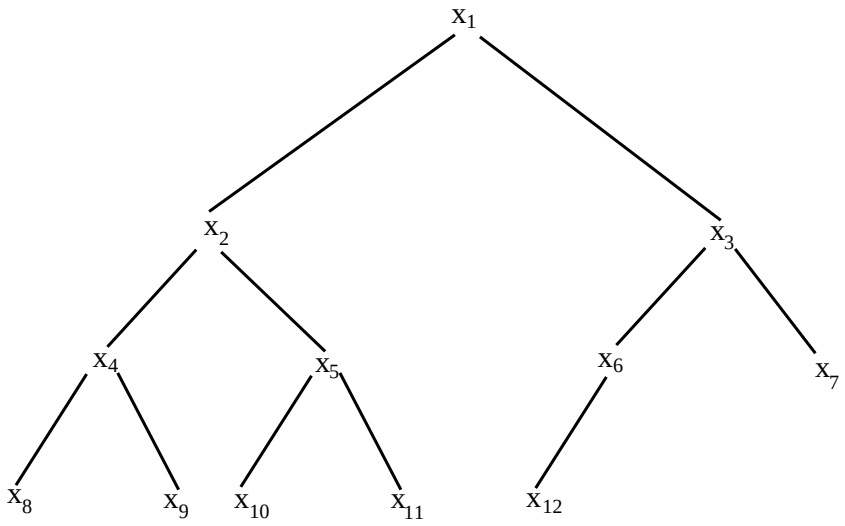
III. AN EFFICIENT ALGORITHM; HEAPSORT

It turns out that there are several algorithms which are *much* more efficient than those described in the previous section. Rather than the operation count being of order N^2 it is of order $N \log_2 N$. Noting, for example, that $\log_2 10^6 \simeq 20$, the gain in efficiency is huge for large N (around $50,000$ for $N = 10^6$), which is surely worth having[1]. *Never* use an N^2 sorting routine for sorting a large data set. This is a massive waste of computer time.

The simplest $N \log_2 N$ routine is "heapsort", for which there is a rather condensed description in Numerical Recipes, Sec. 8.3. Here, I give a considerably expanded explanation.

One starts by putting the data on a **"binary tree",** a graph in which each "node" (i.e. junction) branches into two nodes below it. The elements of the array reside at the nodes, so element x_1 is at node 1 etc. The figure below shows a binary tree for $N = 12$.

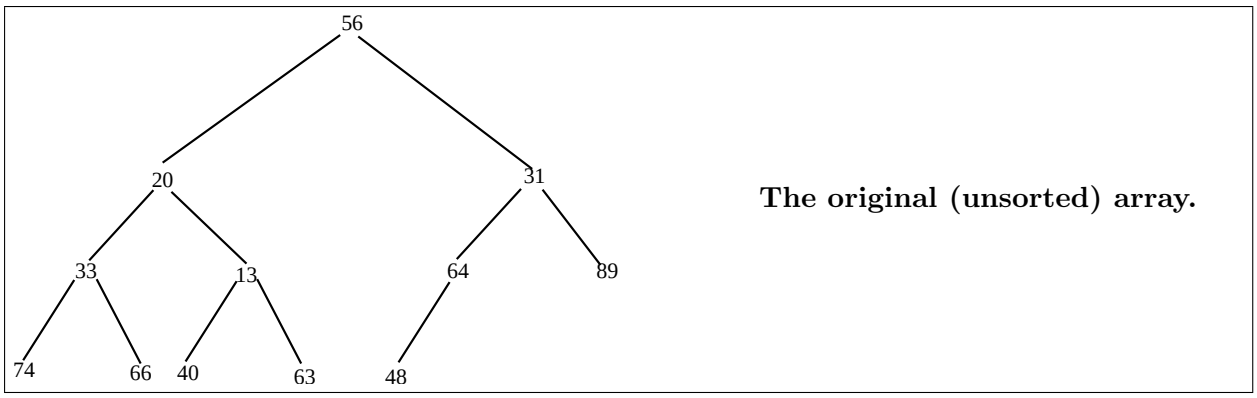
[1] On my laptop I found a time of 8118 secs. for $N = 10^6$ using bubble sort and 0.497 secs. using heapsort, which gives a speedup of about $16,000$. This is 1/3 of the estimated factor of $50,000$ presumably because the heapsort code is a little more complicated. Still a factor of $16,000$ is not too shabby.



Note that unless N is of the form $2^k - 1$ with k a positive integer, the last row will be incomplete, but this will not cause a problem. To go from the top node to one of the bottom nodes needs $\log_2 N$ steps down the tree, rounded down to the next integer. For the above figure this gives 3 steps as observed.

Of course, the data is stored in the computer as a one-dimensional array. However, it is convenient to view it as a binary tree because one can navigate the tree easily. For example, starting from element k, the two elements below it on the tree are $2k$ and $2k+1$, Similarly, the element up the tree is $[k/2]$ where $[k/2]$ means integer divide, i.e. if $k/2$ is not an integer we round down to the closest integer (this is done automatically by integer divide on the computer.) We assume here that the labeling of the elements starts at 1 not 0. (Starting at 1 seems particularly convenient here, sorry C programmers! When programming in C, I suggest that you declare the array containing the values to have length $N+1$, use elements 1 through N, and ignore the '0' element.)

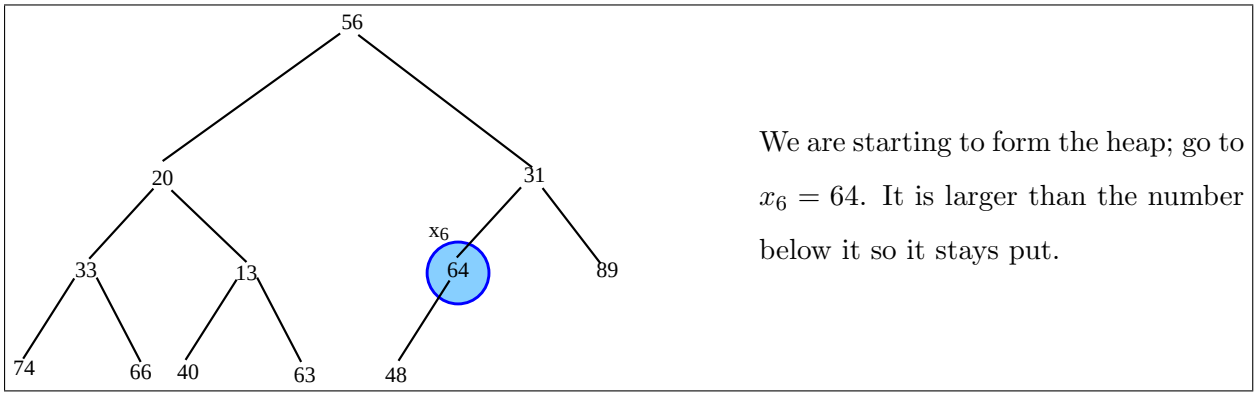
We will sort the following example in which 12 random integers have been generated and put in the array x_i,



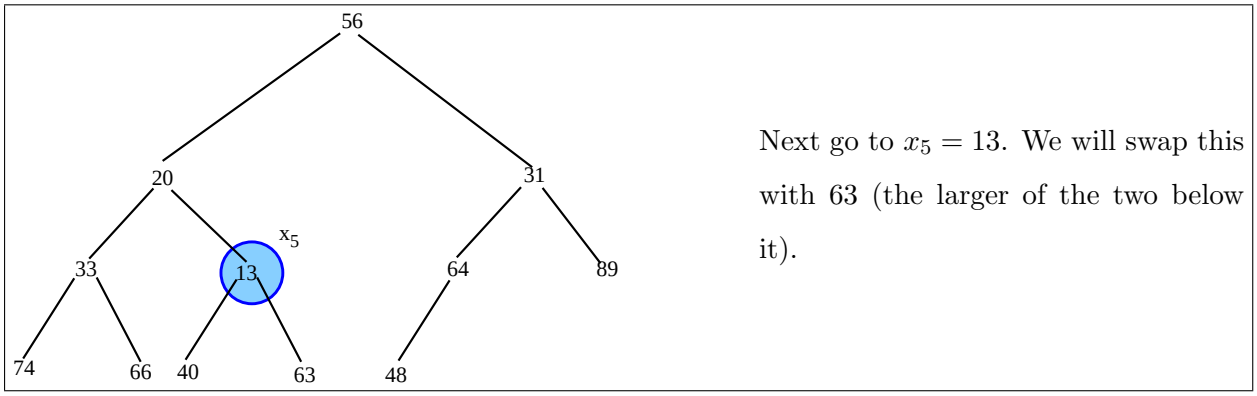
so initially $x_1 = 56, x_2 = 20, x_3 = 32, \cdots, x_{12} = 48$.

The heapsort algorithm proceeds in two stages:

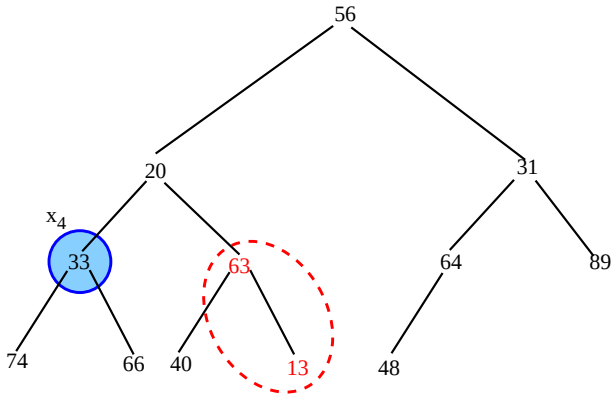
The first stage is to create a "heap", in which each element is larger than both the elements below it. A heap is partially ordered in the sense that there is top to bottom ordering but no ordering from left to right. To create the heap we start with element $[N/2]$. Here $[N/2] = 6$ and $x_6 = 64$. If this element is smaller than at least one of the elements below it we interchange it with the larger of the two elements below. We then see if this element, in its new position, is smaller than at least one of the two below it, and, if so, swap with the larger one. We continue until element $x_{[N/2]}$ has "sifted" down as far as it will go. This sifting procedure is then performed for the previous elements, $x_{[N/2]-1}, x_{[N/2]-2}, \cdots, 2, 1$. At the end, the elements are arranged in a heap. The following set of figures show the creation of the heap for our example above.



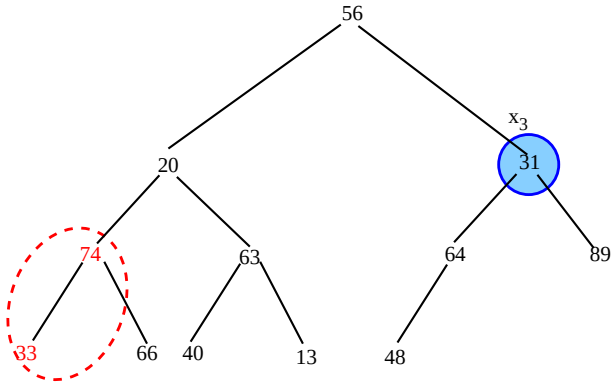
We are starting to form the heap; go to $x_6 = 64$. It is larger than the number below it so it stays put.



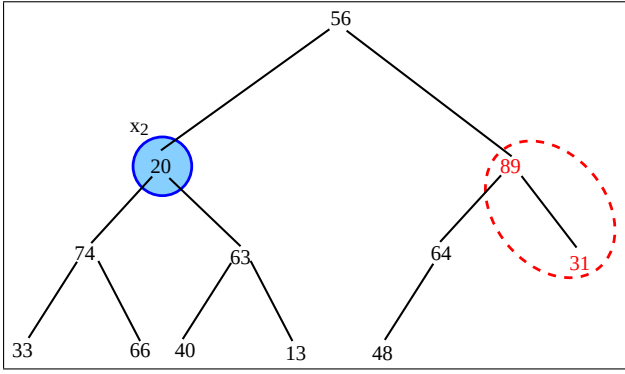
Next go to $x_5 = 13$. We will swap this with 63 (the larger of the two below it).



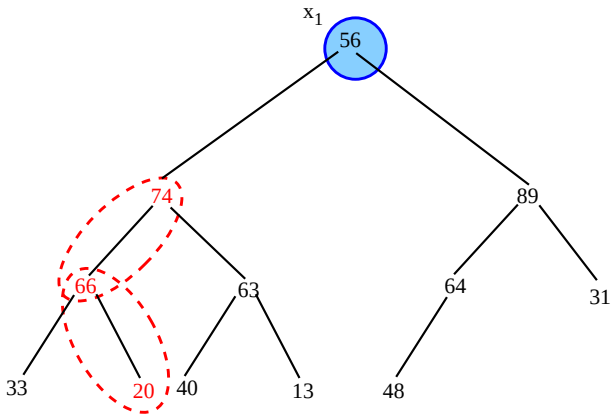
The dashed oval indicates that 13 and 63 have been swapped. The solid circle indicates that we now go to $x_4 = 33$. We will swap this with 74 below it.



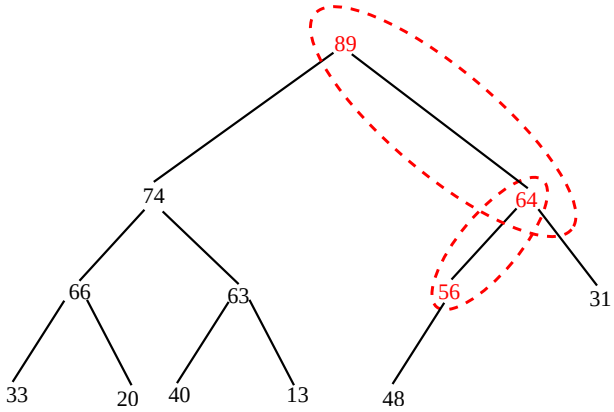
74 and 33 have been swapped. We now go to $x_3 = 31$. This will be swapped with 89.



89 and 31 have been swapped. We now go to $x_2 = 20$. We will swap this with 74 (the larger of the two below it) and then with 66 (the larger of the two below its new position).



20 has been pulled down to the bottom raising up 66 and 74. We now go to $x_1 = 56$. This will be swapped with 89 and then with 64. (It does not continue down to the bottom row because $56 > 48$).



56 moved down two rows, and 64 and 89 moved up.

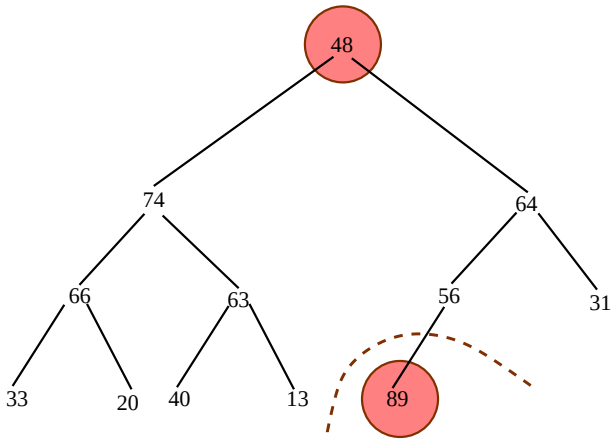
The heap has now been formed.

In the heap there is ordering from top to bottom but not from left to right.

Now that the heap has been formed each element is now greater than the elements below it. Consequently the top element is the largest.

In the second part of heapsort we complete the sorting by repeating the following process $N - 1$ times: we "pull off" the top element and put it in the last "active" place (which initially is just the last element, ($i = N$ (= 12) here), simultaneously putting the element that was in the last active place at the top. We then make the last active element "inactive" by not permitting subsequent "sifting" operations on it.

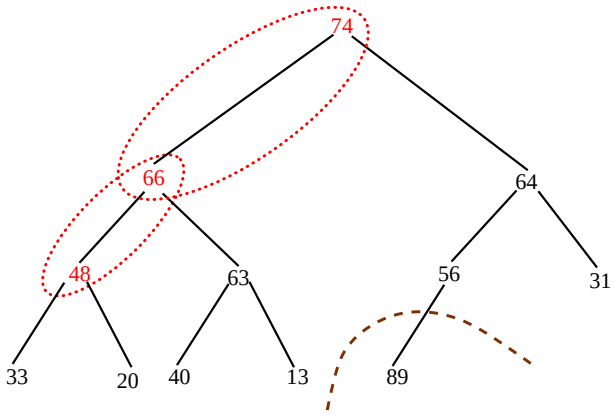
For the first application of this process, all elements are active, the top element has value 89 and this is swapped with the element in the last element, which has value 48. The last node has value 89 (the largest value, so it is its the correct place in the array) is then made inactive, so subsequent "sifting" operations will not be permitted on it. This is illustrated in the next figure.



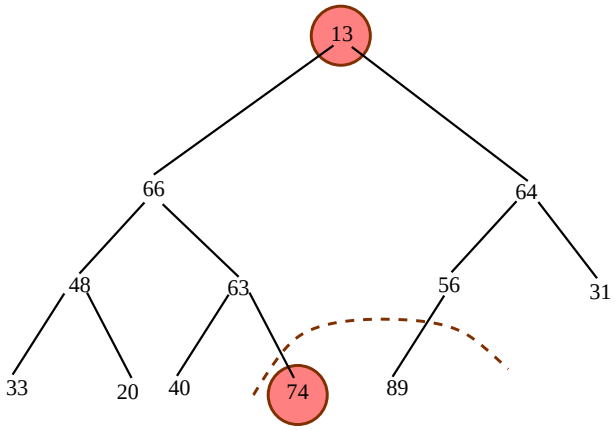
The top element in the previous figure (89) (the largest element) is interchanged with the element in the last node in the previous figure (48). The last node, now containing the largest element, is then made inactive.

The region below the dashed line in this and subsequent figures indicates the "inactive" nodes which will not be altered in subsequent sift down operations. The top element is then sifted down

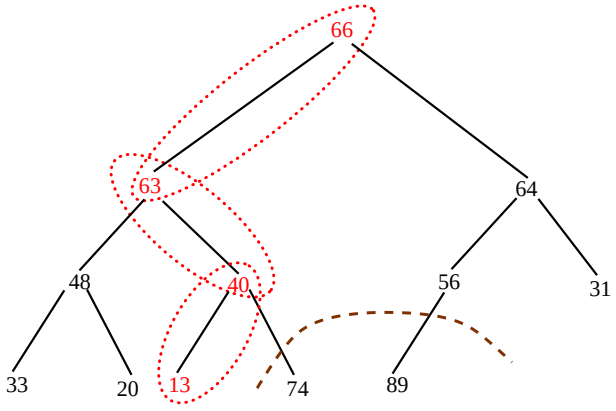
to its right place, which again generates a heap with the largest element on the top. This is then pulled off the top and placed in the last active element which is then made inactive. The following figures show the first few stages in this procedure:



The top element in the previous figure (48) has been sifted down to its correct location, thus raising up 66 and 74. The latter is now the largest active element and so is on top.

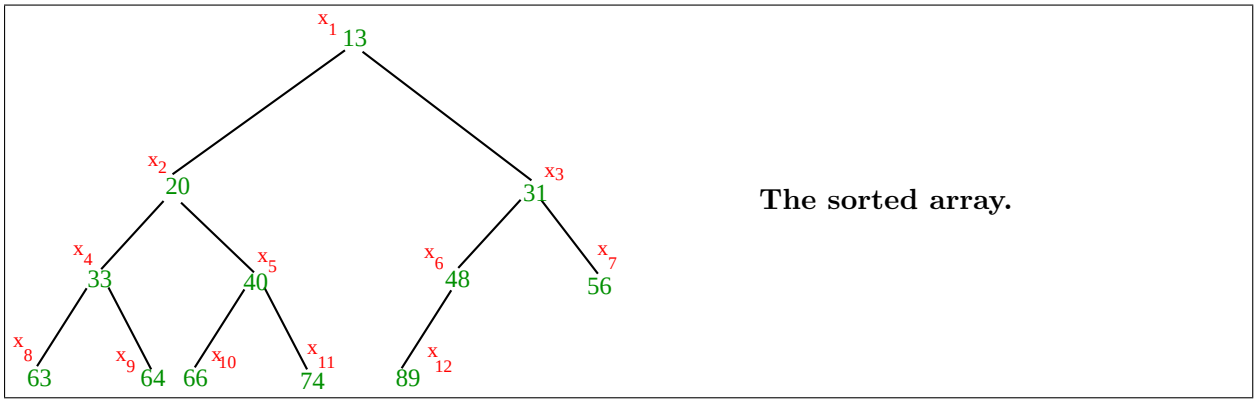


The top element in the previous figure (74) (which is the largest active element) is interchanged with the element in the last active node in the previous figure (13). This last node, now containing the second largest element, is then made inactive.



The top element in the previous figure (13) has been sifted down to its correct location, and the largest active element (66) is thereby brought to the top.

This procedure is repeated 9 more times at which point all the elements have been promoted to the top and sifted down to their correct place. The top element is then the smallest one, and the array is in the correct (ascending) order, with $x_1 = 13, x_2 = 20, \cdots, x_{12} = 89$:



The algorithm involves going through elements and sifting them down to the correct row. Since there are N elements and $\log_2 N$ rows, the operation count is of order $N \log_2 N$, as claimed.

In summary, there are two stages to the process.

1. In the first stage one forms the heap. Elements $[N/2], [N/2]-1, \cdots 2, 1$ are successively sifted down to their right level in the full array. The heap has ordering from top to bottom, but not from side to side.

2. In the second phase, the top element is swapped with the last active element (which is then made inactive) and the new top element sifted down to its correct level among the remaining active elements. This is repeated $N-1$ times, after which the array is in ascending order as required.

Thus, the main ingredient in the code is a function (subroutine) which takes the i-th element of an array of $N_{\rm active}$ elements and sifts it down to the correct level in the tree. In the heap creation phase, i runs over values $[N/2], [N/2]-1, \cdots 2, 1$ while $N_{\rm active}$ is fixed to be N, and, in the second phase, i is fixed to be 1 while $N_{\rm active}$ runs through values $N, N-1, N-2, \cdots, 3, 2$.

As a reminder, it is quite easy to go down the levels of a binary tree because, from node k, the nodes below it are simply $2k$ and $2k+1$ and the node above it is $[k/2]$. Hence the coding is quite simple and efficient.

IV. OTHER ANALOGOUS PROBLEMS

There are other problems in which the operation count is naively N^2 but which can be reduced to $N \log_2 N$ by a clever algorithm. The most famous of these is the Fast Fourier Transform (FFT).

In a discrete Fourier transform (FT) we have values for some quantity f at N evenly spaced values of a parameter t, i.e. we are given $f(t_n)$ where

$$t_n = nh \quad (n = 1, 2, \cdots, N)\,. \tag{1}$$

The Fourier transform $g(\omega_m)$ is defined by

$$g(\omega_m) = \frac{1}{N} \sum_{n=1}^{N} \exp(i\omega_m t_n)\, f(t_n) \tag{2}$$

where

$$\omega_m = \frac{2\pi m}{Nh} \quad (m = 1, 2, \cdots, N)\,, \tag{3}$$

so the N frequencies ω_m are also equally spaced. There is also an inverse transform

$$f(t_n) = \sum_{m=1}^{N} \exp(-i\omega_m t_n)\, g(\omega_m). \tag{4}$$

Equations (2) and (4) are the usual expressions for Fourier transforms except that t and ω have been discretized, which is necessary for numerical work. Often t is time and ω the angular frequency (a "time" Fourier transform), but "spatial" Fourier transforms, in which t is position (and would usually be called x) and ω is wavevector (and would normally be called k), are also commonly performed.

Doing the FT in Eq. (2) naively requires N^2 operations, since there are N terms in the sum which have to be carried out for each of the N values of m. However, by carefully grouping terms together, the FFT performs the calculation in $N \log_2 N$ operations. Unfortunately, time does not permit us to describe this important technique, and the interested student is referred to the books, such as Numerical Recipes.