

Sorting isn't only relevant to records and databases. It turns up as a component in algorithms in many other application areas surprisingly often.

There are many graph algorithms which include sorting as a crucial component, for example ones to solve the problem of finding a **minimal spanning tree**.

Let  $G = \langle N, E \rangle$  be a **connected, undirected** graph where  $N$  is the set of **nodes** and  $E$  is the set of **edges**.

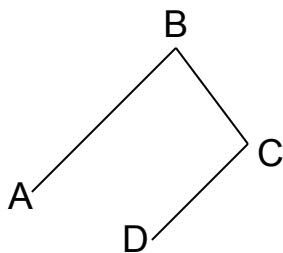
If the number of nodes is  $n$ , and the number of edges is  $e$ , it is always the case that

$$n - 1 \leq e \leq \frac{n}{2}(n - 1)$$

↑  
connected graph  
with no cycles

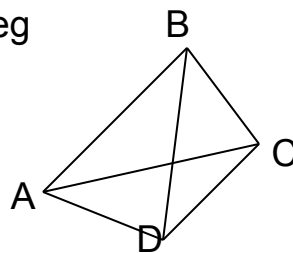
↑  
every node is connected  
to every other

eg



$E = [ [A,B], [B,C], [C,D] ]$

eg



$E = [ [A,B], [A,C], [A,D], [B,C], [B,D], [C,D] ]$

A **minimal spanning tree** is a subset  $T \subseteq E$  such that

- a) All the nodes remain connected when only the edges in  $T$  are used ('spanning' property)
- b) The sum of the lengths (weights) of the edges in  $T$  is as small as possible (it is 'minimal').  
→  $\#T = n-1$  (no cycles).

## Kruskal's algorithm

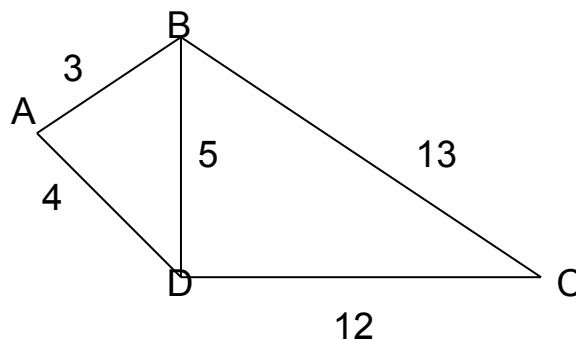
The algorithm has the following schematic form:

- (1) Sort the set of edges  $E$  by increasing edge length
- (2)  $T \leftarrow []$  (*will ultimately contain the edges of the minimal spanning tree*)
- (3) Initialise  $n$  sets, each containing 1 node of  $N$
- (4) repeat

```
[U,V] ← shortest edge not yet considered
U_set (set containing node U) ← find (U)
V_set (set containing node V) ← find (V)
If U_set ≠ V_set
merge (U_set,V_set) ← only add an edge if it joins 2 nodes
                        in different connected components
T ← T + [ [U,V] ]
```

until (number of edges in  $T = n-1$ )

For example consider the graph



$N = [A, B, C, D]$ ,  $E = [ [A,B], [A,D], [B,D], [B,C], [D,C] ]$   
 $n = 4$ ,  $e = 5$

<u>Step</u>	<u>Edge considered</u>	<u>Connected components</u>
Initialisation	-	[A] [B] [C] [D]
1	[A, B]	<p>T= [ [A,B] ]</p>
2	[A,D]	<p>T= [ [A,B],[A,D] ]</p>
3	[B,D]	rejected
4	[D,C]	<p>T= [ [A,B],[A,D],[D,C] ]</p>

At this point the number of edges in the connected subset T is equal to 4 ( $n-1$  for this 5-node graph) and Kruskal's algorithm terminates.

## Complexity of Kruskal's algorithm

(1) Sorting the edges (eg with MergeSort) should take time in

$$O(e \log e) = O(e \log n)$$

$$\text{(since } n - 1 \leq e \leq \frac{n}{2}(n - 1)$$

$$\log(n) \leq \log e \leq 2 \log n \text{ as } n \rightarrow \infty)$$

(2) Initialise set T to empty set: **O(1)**

(3) Initialise n disjoint sets: **O(n)**

(4) Number of **merge operations** = number of edge additions to T  
= **n-1**.

In the worst case, all the edges have to be considered. Each candidate edge requires two find operations (one for each end) so in the worst case there are **2e find operations** altogether.

It can be shown (Brassard and Bratley pp 60-63) that the cost of either a find or merge operation is given by

$$\lg^*(n) \equiv \min \{k \mid \underbrace{\lg \lg \lg \dots \lg n}_{k \text{ times}} \leq 0\}$$

↑  
k times

$$\text{( eg } n=4 = 2^2$$

$$\lg 4 = 2 = 2^1$$

$$\lg \lg 4 = 1 = 2^0$$

$$\lg \lg \lg 4 = 0 \rightarrow k=3, \text{ and so } \lg^*(4) = 3)$$

$\lg^*(n)$  is a function which grows very, very slowly:

$$\lg^*(n) \leq 5 \text{ for } n \leq 65,536 (2^{16}); \lg^*(n) \leq 6 \text{ for } n \leq 2^{65,536}$$

It is for all practical purposes constant (  $O(1)$  )

$$\text{Therefore Step (4)} \in O( (2e+n-1) \lg^*(n) ) \approx O(e)$$

**Thus the complexity of Kruskal's algorithm is given by that of the initial sorting process and is in  $O(e \log n)$ .**

The efficiency of Kruskal's algorithm in practice will depend on how densely the graph is connected. For a sparse graph with a close-to-linear number of edges Kruskal's algorithm will be in  $O(n \log n)$ , but for a more strongly connected graph (where the number of edges tends toward a quadratic function) an alternative such as **Prim's algorithm** may be preferable.

## DECISION TREES AND LOWER BOUNDS

Many algorithms, including all commonly used searching and sorting algorithms, proceed via a sequence of comparisons.

The operation of such algorithms can be represented by a **decision tree** in which the flow of control of the algorithm commences at the **root** and proceeds eventually to a **leaf**, at which point it delivers an output. Each node represents a comparison (more generally, a question asked) in the course of operation of the algorithm that has outcomes determining the subsequent actions of the algorithm from that point.

The **depth** of the decision tree,  $d$ , is the distance from the root to the furthest leaf. It represents the number of comparisons that must be carried out by the algorithm in the worst case.

We will assume for simplicity the questions asked have only two possible answers, **yes** (Y) or **no** (N) (**binary decision tree**).

Because in this case there are only two branches (Y/N) from each node the number of nodes on each level of such a tree cannot be more than twice the number on the preceding level, and hence for a binary decision tree of depth  $d$ , **number of leaves**  $\leq 2^d$ .

## Decision tree example: guessing game

You are asked to guess, using a minimum of questions with yes/no answers, a number from  $0..n-1$ .

What is the best strategy?

One obvious approach would be to ask

"Is it 0?"

"Is it 1?"

...

"Is it  $n-3$ ?"

"Is it  $n-2$ ?"

or the equivalent sequence of questions

"Is it less than 1?"

"Is it less than 2?"

...

"Is it less than  $n-2$ ?"

"Is it less than  $n-1$ ?"

where in either case a 'yes' to the last question means the number is  $n-2$ , a 'no' that it is by default -- we assume the game is played fairly -- equal to  $n-1$ .

In the **worst case** (when the number is either  $n-2$  or  $n-1$ ) this would require  **$n-1$**  questions to be asked.

On **average** the number of questions that are asked is

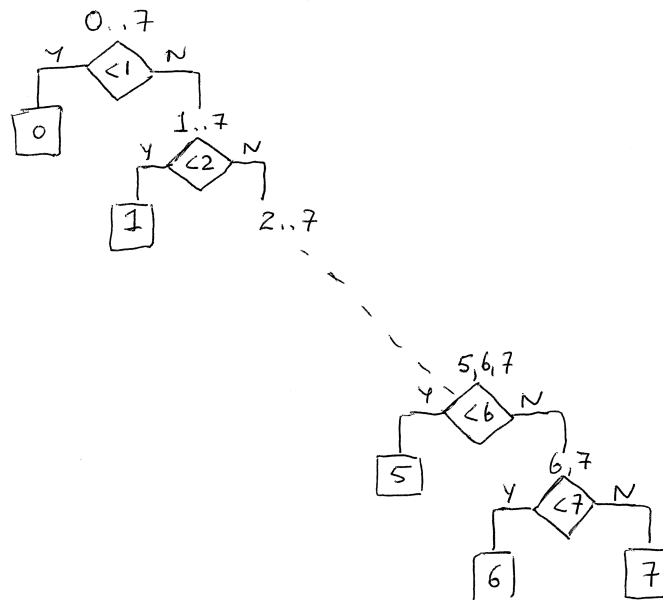
$$\frac{1}{n} \sum_{i=0}^{n-2} (i+1) + \frac{n-1}{n}$$

where the  $1/n$  weights all the possibilities from  $0..n-1$  equally, the sum covers the questions asked if the number turns out to be  $0..n-2$  and the contribution  $(n-1)/n$  is for the possibility the number is  $n-1$  (also discovered by asking the maximum of  $n-1$  questions).

This average can be seen to be equal to

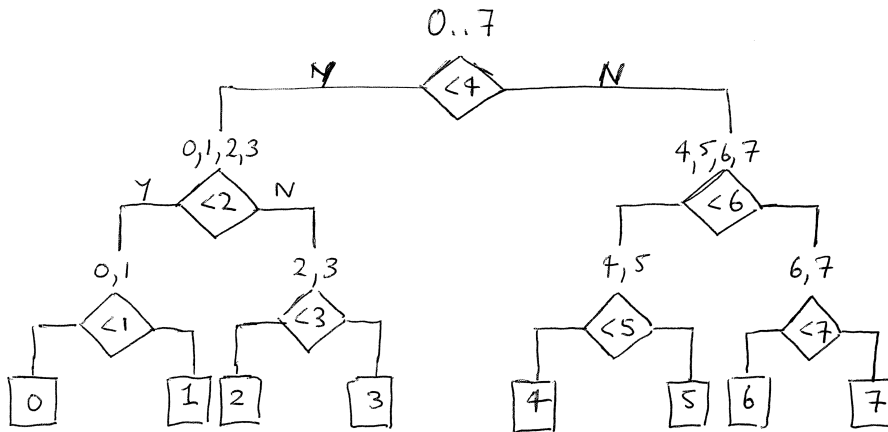
$$\frac{1}{n} \sum_{i=1}^{n-1} i + \frac{n-1}{n} = \frac{1}{2}(n-1) + \frac{n-1}{n} = \underline{\underline{\frac{1}{2}(n+1) - \frac{1}{n}}}$$

A decision tree for this not-too-bright algorithm, for the case that  $n=8$  ('guess a given number in the range  $0..7$ ') is given below:



(The numbers above the decision nodes correspond to the possibilities remaining before the next question is asked.)

The decision tree below shows a much better way to guess a number in the range 0..7:



Here every case needs the same number of questions to be asked, three.

Guessing any number 0..n-1 by this method requires  $\lceil \log_2 n \rceil$  questions to be asked.

---

Suppose for numbers in the range 0..7 it is the number 5 that is to be guessed.

The decision tree proceeds from its root to the leaf '5' by the route

```
"Is it less than 4? N
"Is it less than 6? Y
"Is it less than 5? N
```

This is equivalent to the three questions (starting in ignorance of all three of the bit-values, state \* \* \* )

```
"Is the 1st bit 0?" N 1 * *
"Is the 2nd bit 0?" Y 1 0 *
"Is the 3rd bit 0?" N 1 0 1
```

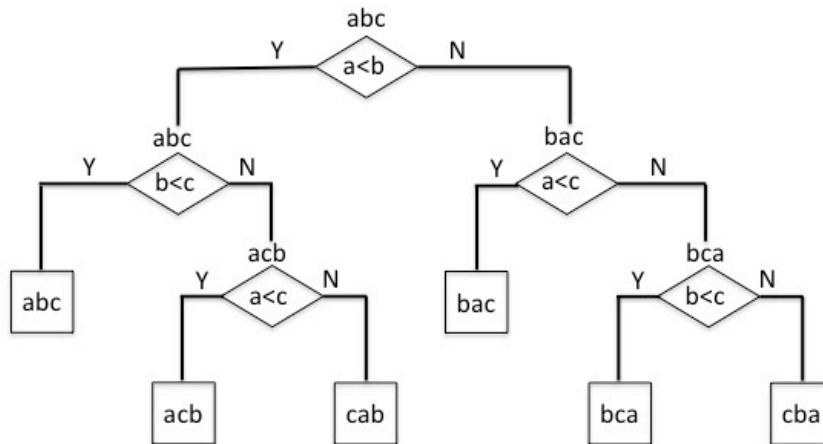
Establishing the value of the number to be guessed takes  $\lceil \log_2 n \rceil$  questions, an example of an **information-theoretic** lower bound.

In this approach one attempts to establish a lower bound on the amount of work that must be done by any algorithm in its worst case by looking at the **amount of information** a solution to a problem would have to produce.

No algorithm for the guessing game can ask fewer questions and hence the one whose decision tree exactly mimics this sequence of questions has to be optimal for this problem.

## Lower bound for comparison-based sorting

The decision tree below shows the flow of control when InsertionSorting an array with initial contents [a,b,c]:



(The sequences above the decision nodes correspond to the array ordering before the next comparison is carried out.)

Notice there are  $6=3!$  leaves, equal to the number of possible orderings of the three numbers a, b, c.

In general a decision tree for a comparison-based sort of n elements must have **at least n! leaves** since there must be at least one leaf corresponding to each possible ordering of the elements.

But we know the number of leaves can't be more than  $2^d$ , where d is the depth of the tree, equal also to  $C_{\text{worst}}$ , the worst-case cost in terms of the number of comparisons.

Hence

$$2^d \geq \text{number of leaves} \geq n!$$

$$\rightarrow 2^d \geq n!$$

and by taking the log of both sides it can be seen that

$$C_{\text{worst}} \geq \log_2(n!)$$

Since it can be shown that  $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$  for large  $n$  (Stirling's formula) it follows that

$$\begin{aligned} \log_2(n!) &\approx n \log_2 n - n \log_2 e + \frac{1}{2} \log_2 n + \dots \\ &\approx n \log_2 n - 1.44n \end{aligned}$$

and hence that the 'best worst case' cost

$$C_{\text{worst}} \in O(n \log n)$$

**-- any comparison-based sorting algorithm cannot use less than a multiple of  $n \log n$  comparisons in its worst case.**

(It can also be seen that Mergesort, with a time-demand  $\approx n \log_2 n - n$ , is close to optimal in worst-case performance.)