Why time complexity is important

The time complexity of an algorithm can make a big difference as to whether it is practical to apply it to large instances.

Suppose the most frequent operations take $1\,\mathrm{ns}$

	$n = 10$	$n = 50$	$n = 100$	$n = 1000$
$\log_2 n$	$3\,\mathrm{ns}$	$5\,\mathrm{ns}$	$6\,\mathrm{ns}$	$10\,\mathrm{ns}$
n	$10\,\mathrm{ns}$	$50\,\mathrm{ns}$	$100\,\mathrm{ns}$	$1\,\mu\mathrm{s}$
$n \log_2 n$	$33\,\mathrm{ns}$	$282\,\mathrm{ns}$	$664\,\mathrm{ns}$	$10\,\mu\mathrm{s}$
n^2	100 ns	$2.5\,\mu\mathrm{s}$	10 $\mu\mathrm{s}$	1 ms
n^3	1 $\mu\mathrm{s}$	$125\,\mu\mathrm{s}$	1 ms	1 s
n^{100}	3×10^{83}y	2.5×10^{179}y	3×10^{209}y	3×10^{310}y
1.1^n	$2.6\,\mathrm{ns}$	$117\,\mathrm{ns}$	$13\,\mu\mathrm{s}$	8×10^{50}y
2^n	$1\,\mu\mathrm{s}$	$3.\,5 \times 10^{24}$y	4×10^{39}y	3×10^{310}y
$n!$	$3\,\mathrm{ms}$	10×10^{73}y	3×10^{167}y	1.3×10^{2577}y
2^{2^n}	6×10^{317}y	big	Bigger	HUGE

Another way to look at it is how big an instance can be solved in a given time

	In 1s	In 1hr	In 1day
$\log_2 n$	$10^{300,000,000}$	big	really big
n	10^9	3.6×10^{12}	8.64×10^{13}
$n \log_2 n$	4×10^7	10^{11}	2×10^{12}
n^2	3×10^4	2×10^6	9×10^6
n^3	1000	$15,000$	$44,000$
2^n	29	42	46
$n!$	12	15	16
2^{2^n}	4	5	5

We broadly classify functions according to how they behave

- Super exponential functions grow faster than any exponential function.
  - $n!$, 2^{2^n}, The Ackermann/Peter function.
- Exponential functions
  - 2^n, 3^n etc
  - $2^{\Theta(1)n}$ where $\Theta(1)$ represents some positive coefficient
  - (The next time someone refers to “exponential growth” ask yourself if they know what they are talking about.)
- Polynomial functions n , $n \log_2 n$, n^2, n^3 etc
  - While $n \log n$ is not really a polynomial, it is bounded by two polynomials and so is considered a polynomial.
  - $n^{\Theta(1)}$ where $\Theta(1)$ represents some positive coefficient

- Polylog functions (polynomials of logs).
  - $\log_2 n$, $(\log_2 n)^2$, ...
  - $(\log n)^{\Theta(1)}$

Feasible and infeasible

As a general rule, we say that any algorithm that is

- polynomial time or better is **feasible**
- superpolynomial time is **infeasible**

However, as you can see from the tables, if you are dealing with billions of bits then even an n^2 algorithm is impractical.

In compiling, it is a general rule that any 'optimization' method should be $\Theta(n^2)$ or better in the size of a subroutine.

Obviously any $\Theta(n^{100})$ time algorithm will be impractical. In practice polynomial-time algorithms on RAMs are usually not worse than $\Theta(n^6)$, since each increase in the exponent corresponds to some complication in the algorithm such as an additional loop nesting.

One benefit of drawing the line at polynomial time is that the model of computation is not important. E.g., any polynomial RAM algorithm can be translated to a polynomial time algorithm on a Turing machine and conversely.

Upper and lower bounds on function complexity

Suppose we don't know whether or not $f \in \Theta(g)$.

It is possible that still that we know something about their relationship.

Definition: Function f is **asymptotically dominated** by g if and only if there exist positive numbers b, and m such that

$$\forall n \in \mathbb{N} \cdot n > m \Rightarrow f(n) < b \times g(n)$$

Notation: We write $f \preceq g$ to mean that f is **asymptotically dominated** by g.

In terms of sets, we say $f \in O(g)$.

I.e. $O(g)$ is the set of all functions dominated by g.

$$O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n)$$

Definition: Function f **asymptotically dominates** by g if and only if there exist positive numbers a, and m such that

$$\forall n \in \mathbb{N} \cdot n > m \Rightarrow a \times g(n) < f(n)$$

Notation: We write $f \succeq g$ to mean that f **asymptotically dominates** g.

In terms of sets, we say $f \in \Omega(g)$.

I.e. $\Omega(g)$ is the set of all functions that dominate g.

$$\Omega(n) \supseteq \Omega(n \log n) \supseteq \Omega(n^2) \supseteq \Omega(n^3) \supseteq \Omega(2^n)$$

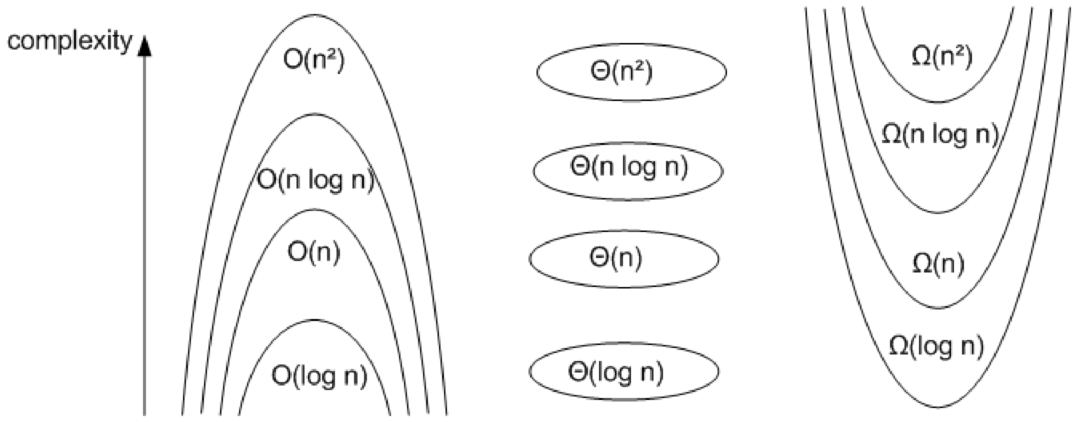
We have $f \preceq g$ if and only if $g \succeq f$; in other words

$f \in O(g)$ if and only if $g \in \Omega(f)$

Relation to Θ. For any g:

- $\Theta(g) \subset O(g)$
- $\Theta(g) \subset \Omega(g)$
- $\Theta(g) = O(g) \cap \Omega(g)$

Comparing O, Θ, and Ω in three Venn diagrams



Problem complexity

A **problem** consists

- an input set (called the set of instances)
- a set of outputs.
- an acceptability relation between these sets
- a measure of input size.

Using precondition P and postcondition R:

- P identifies inputs out of a (possibly) larger set.
- R defines the acceptable outputs for each input.

An algorithm A **solves** problem B iff, for every input, A terminates with an acceptable output.

Exact problem complexity

The **worst-case time complexity of a problem** is the worst-case time complexity of the fastest algorithm that solves it.

For the rest of this slide deck, we are concerned only with worst-case time complexity.

Note that problem complexity is model dependent. Usually the RAM model is considered the standard.

When possible, the complexity of a problem should be stated using $\Theta(f)$ notation.

Upper-bounds

Often the exact problem complexity of a problem is hard to calculate exactly as we must consider every algorithm for the problem — including ones that no one has yet thought of.

You may know a fast algorithm for a problem.

- But that doesn't mean that Fred won't come up with a faster one tomorrow

If you know an algorithm has a worst-case time function in $O(g)$ then the problem complexity is in $O(g)$.

For example,

- merge sort sorts n numbers in $\Theta(n \log n)$ time
- therefore (since $\Theta(g) \subseteq O(g)$) merge sort sorts in $O(n \log n)$ time
- therefore the fastest sorting algorithm sorts in $O(n \log n)$ time
- therefore (by definition) the problem complexity of sorting is in $O(n \log n)$
- in other words $O(n \log n)$ is an upper bound for sorting n numbers.

Proof of the third step: Assume (falsely) that the fastest sorting algorithm does not take $O(n \log n)$ time. Then merge sort is faster that the fastest sorting algorithm. Contradiction.

Note that we can not (yet) conclude that the fastest sorting algorithm sorts in $\Theta(n \log n)$.

Nor can we (yet) conclude that the complexity of sorting is in $\Theta(n \log n)$

When problem complexity is stated using O, we say the result is an upper-bound.

Lower-bounds

What if we can prove that no algorithm can be quicker than $\Theta(f)$ time?

- Then we have that $\Omega(f)$ is a lower-bound for the problem.

Case study — Matrix multiplication

Matrix multiplication (MM). Calculate $A \times B$ where A and B are n by n matrices

From your high-school education you know that we can compute MM with

```
for $i, j \in \{0, ..n\}$
    $C(i, j) := 0$
    for $k \in \{0, ..n\}$ $C(i, j) := C(i, j) + A(i, k) \times B(k, j)$
    end for
end for
```

This is a $\Theta(n^3)$ algorithm and so we know that the time complexity of MM $\in O(n^3)$.

Since any algorithm has to at least look at each input number and there are $2n^2$ of them, any algorithm for MM can not have a complexity of better than $\Omega(n^2)$.

Conclusion:

- The complexity of MM is in $\Omega(n^2) \cap O(n^3)$.

This conclusion is secure against the future.

- Discovery of new algorithms will not invalidate it
- However new ideas may improve on it.

In 1969 Volker Straßen (to much surprise) found a way to multiply matrices that takes $\Theta(n^{\log_2 7})$ (approximately $\Theta(n^{2.807})$) and so we can improve the result.

- The complexity of MM is in $\Omega(n^2) \cap O(n^{\log_2 7})$

In 1990 Coopersmith and Winograd found an algorithm that is in (approx) $O(n^{2.376})$.

And so we can conclude that

- The complexity of MM is in (approx) $\Omega(n^2) \cap O(n^{2.376})$

Recently Le Gall lowered the upper bound to $O(n^{2.3728639})$

It is an open problem whether or not the complexity of MM is in $O(n^2)$ (and hence in $\Theta(n^2)$)

Case study: sorting

As seen above, the complexity of sorting n numbers is upper bounded by $O(n \log n)$.

In this section we will show that:

- In a restricted model of computation, the complexity is lower bounded by $\Omega(n \log n)$
- and so (since merge sort fits into this model), the complexity of sorting (in the restricted model) is $\Theta(n \log n)$

A restricted model of computation

In this model of computation, two items may be compared but we will not otherwise access the value of an item, other than to copy it

An algorithm in this model can be thought of as a set of trees, one for each tree for size of input.

Each tree node either represents an action or a comparison. Action nodes have zero or one child, comparison nodes have 1 or 2 children. (A comparison whose conclusion is forgone has 1 child.)

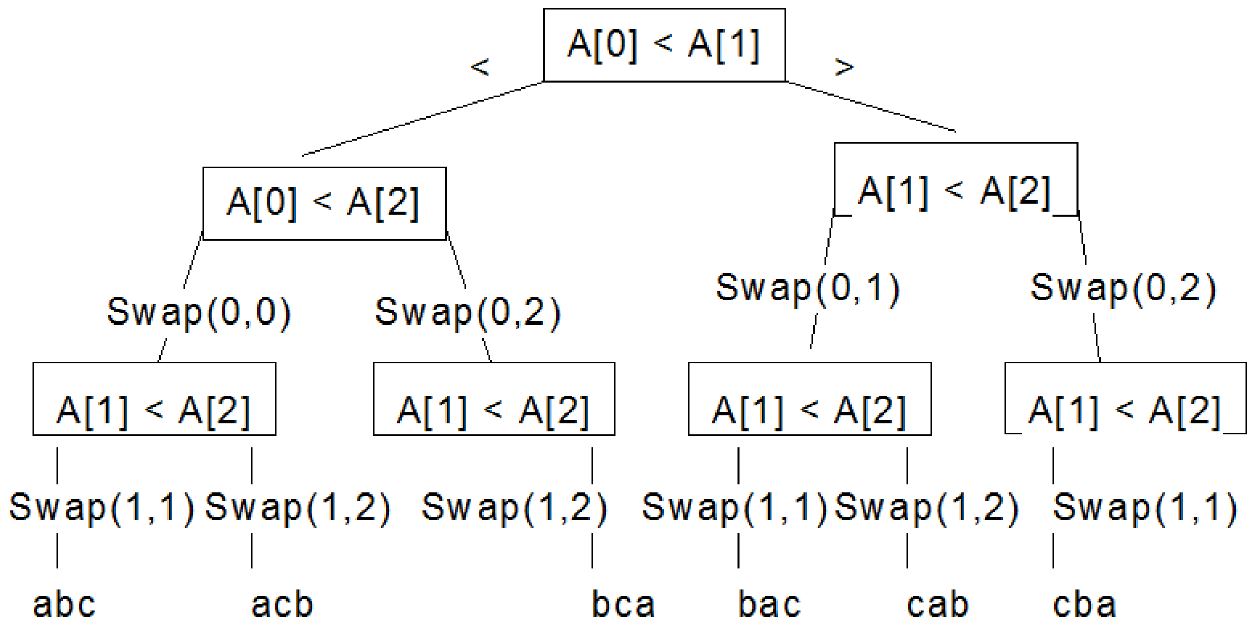
For simplicity, I'll assume that no two items of the input are the same.

[Considering only a restricted set of inputs is kosher, because any lower bound we find for this restricted problem must also hold for the unrestricted problem.]

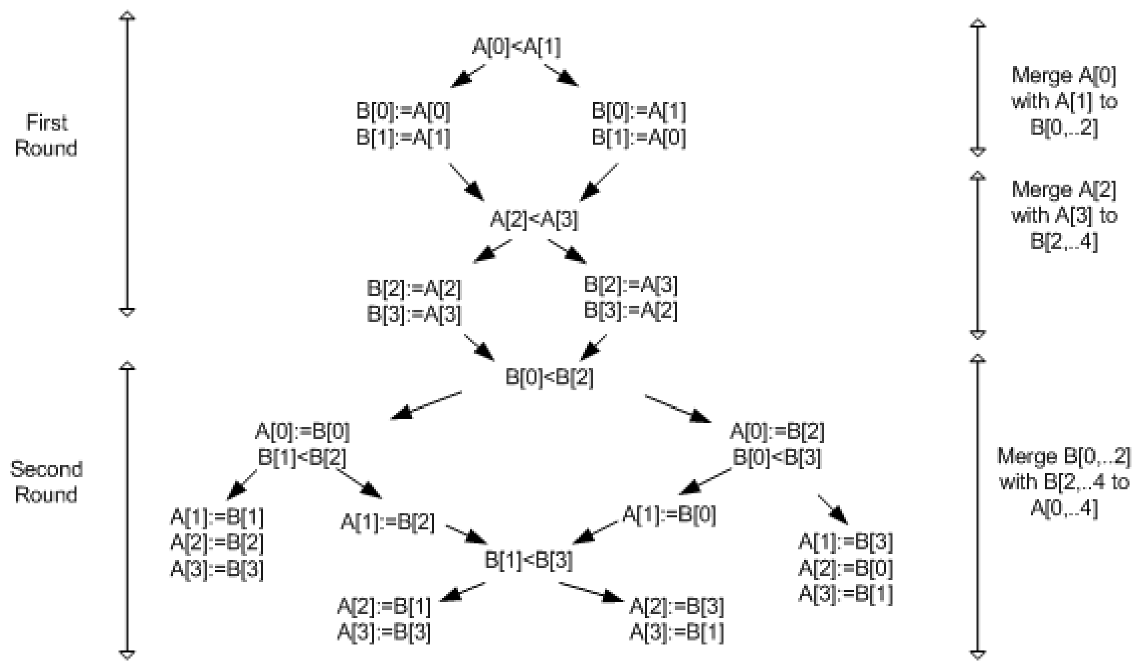
Selection sort

```
for $i \leftarrow [0, ..n-1]$ do
    var $j := i$
    for $k \leftarrow [i+1, ..n]$ do if( $a[k] < a[j]$ then $j := k$ end if
    end for
    swap$(i, j)$
end for
```

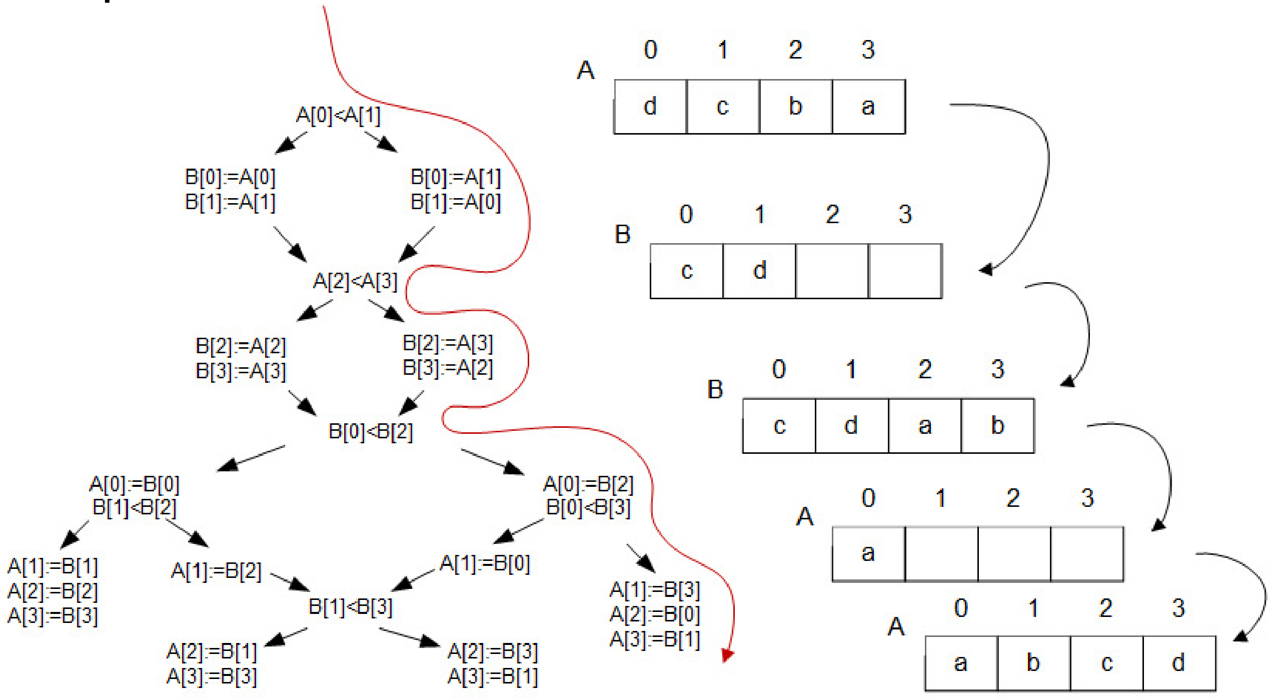
Here is a tree for selection sort with $n = 3$. Assume that $a < b < c$ the leaves show the input that corresponds to each path.



Next is a directed acyclic graph that can be expanded to a tree with 24 leaves representing merge sort for $n = 4$


First Round
Second Round
A[0]<A[1]
B[0]:=A[0] B[1]:=A[1]
B[0]:=A[1] B[1]:=A[0]
A[2]<A[3]
B[2]:=A[2] B[3]:=A[3]
B[2]:=A[3] B[3]:=A[2]
B[0]<B[2]
A[0]:=B[0] B[1]<B[2]
A[0]:=B[2] B[0]<B[3]
A[1]:=B[1] A[2]:=B[2] A[3]:=B[3]
A[1]:=B[2]
A[1]:=B[0]
A[1]:=B[3] A[2]:=B[0] A[3]:=B[1]
B[1]<B[3]
A[2]:=B[1] A[3]:=B[3]
A[2]:=B[3] A[3]:=B[1]
Merge A[0] with A[1] to B[0,..2]
Merge A[2] with A[3] to B[2,..4]
Merge B[0,..2] with B[2,..4 to A[0,..4]

Example:


A[0]<A[1]
B[0]:=A[0] B[1]:=A[1]
B[0]:=A[1] B[1]:=A[0]
A[2]<A[3]
B[2]:=A[2] B[3]:=A[3]
B[2]:=A[3] B[3]:=A[2]
B[0]<B[2]
A[0]:=B[0] B[1]<B[2]
A[0]:=B[2] B[0]<B[3]
A[1]:=B[1] A[2]:=B[2] A[3]:=B[3]
A[1]:=B[2]
A[1]:=B[0]
A[1]:=B[3] A[2]:=B[0] A[3]:=B[1]
B[1]<B[3]
A[2]:=B[1] A[3]:=B[3]
A[2]:=B[3] A[3]:=B[1]
A 0 1 2 3 d c b a
B 0 1 2 3 c d
B 0 1 2 3 c d a b
A 0 1 2 3 a
A 0 1 2 3 a b c d

For a given n

- the best case time is the length of the shortest path from root to leaf
- the worst case time is the length of the longest path from root to leaf

To simplify, we'll only count comparisons.

[Since we are investigating lower bounds, it is kosher to ignore whole classes of operations. If a sorting algorithm requires at least $f(n)$ comparisons, then it must require at least $f(n)$ operations.]

Thus the worst (and best, and average) case time for selection sort is $\frac{n^2-n}{2}$ comparisons.

But selection sort is not the best algorithm. We want a result that even the best algorithm can not beat.

Merge sort, for $n = 2^k$, requires $\Theta(n \log n)$ comparisons. Can we do better?

The best algorithm has the shortest trees (as n approaches ∞)

The key question is:

- *How short can a tree for an input of size n be?*

The inputs $[2, 1, 3]$ and $[4, 1, 6]$ look the same in this model, as the only way to access the data is by comparing.

There are $n!$ distinct inputs, as there are $n!$ permutations of n distinct values.

Any two permutations of the input require the algorithm to do something different. Each possible input requires a different path through the tree and hence a different leaf.

There are $n!$ leaves in each tree, for inputs of size n, *regardless of the algorithm*.

So our key question becomes

- *If a binary tree has $n!$ leaves, what is the shortest its longest path can be?*

We need to consider the squattiest trees.

Lemma 0 a binary tree of height x has at most 2^x leaves.

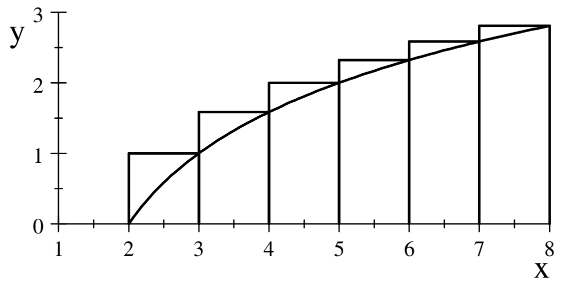
Corollary 0 a binary tree with y leaves has height at least $\lceil \log_2 y \rceil$.

Lemma 1 $\log_2(n!) \in \Omega(n \log n)$

Proof of lemma 1:

$$
\begin{array}{ll}
 & \log_2(n!) \\
= & \\
 & \log_2(1 \times 2 \times ...n) \\
= & \\
 & \log_2 2 + \log_2 3 + ... + \log_2 n \\
> & \\
 & \displaystyle\int_2^{n+1} \log_2 (x-1) \; dx
\end{array}
$$

To justify the last step, I provide a “proof by picture”:



$$\log_2 2 + \log_2 3 ... + \log_2 n > \int_2^{n+1} \log_2 (x-1)\ dx$$

$$\int_2^{n+1} \log_2 (x-1)\ \mathrm{d}x$$

$=$

$$\frac{1}{\ln 2} \int_1^n \ln x\ \mathrm{d}x$$

$=$ “$\int \ln x \, \mathrm{d}x = x \ln x - x$”

$$\frac{1}{\ln 2} ((n \ln n - n) - (1 \ln 1 - 1))$$

$=$

$$\frac{1}{\ln 2} (n \ln n - n + 1)$$

$\in$ “The dominant term is $\frac{1}{\ln 2} n \ln n$”

$$\Omega(n \log n)$$

Proof of the main result

$$\begin{array}{ll} & \text{The height of a binary tree with } n! \text{ leaves} \\ \geq & \qquad \text{"Corollary 0"} \\ & \lceil \log_2 n! \rceil \\ \in & \qquad \text{"Lemma 1"} \\ & \Omega(n \log n) \end{array}$$

Therefore, for every algorithm, there is at least one input that requires at least $\Omega(n \log n)$ comparisons to sort.

So $\Omega(n \log n)$ is a lower bound on the complexity of sorting.

- There is no point trying to find an algorithm that is significantly better than merge sort or heap sort (that accesses the data only by comparison).

Since $O(n \log n)$ is an upper bound and $\Omega(n \log n)$ is a lower bound, $\Theta(n \log n)$ is an exact bound.

We now know the complexity of sorting.

How good is merge sort quantitatively?

How many comparisons does merge-sort use? For n a power of 2 we have (worst-case)

$$\begin{aligned} m(1) &= 0 \\ m(2^k) &= 2m(2^{k-1}) + 2^k - 1 \end{aligned}$$

So

$$m(1024) = 9217$$

For comparison

$$\lceil \log_2(1024!) \rceil = 8770$$

So merge sort is quite close to optimal.

(The reason quicksort is usually quicker than merge-sort is that it has a smaller number of moves.)

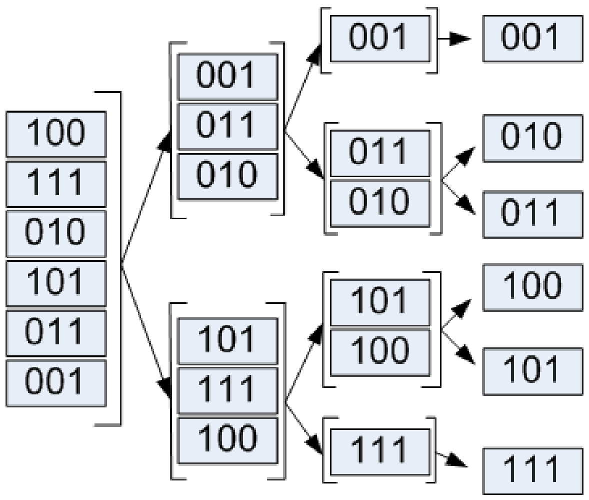
Sorting in $\Theta(n)$ time

For this section we will assume that the items to be sorted are natural numbers and are no bigger than some constant 2^m.

If it is not known a priori, a preliminary pass can be made to determine m.

Radix sorting msb to lsb

Radix sorting is a lot like quick sort. Look at the most-significant bit position of each number. Move the numbers with 0 at this bit position to the front of the array. Now do a radix sort of each 'half', only this time, using the next-most-significant bit and so on down until all bits are taken care of.



This takes $\Theta(mn)$ time (and can be very fast in practice).

This is linear in several senses

- For each particular m, we have $\Theta(n)$. E.g. sorting n 32-bit numbers can be done in $\Theta(n)$.
- If m is not fixed, then a realistic measure of the input

size is not numbers (n), but rather bits (mn).

- A straight-forward generalization to variable sized data (e.g. variable length strings in lexicographic order) can be done in time $\Theta(N + n)$ where N is the total number of characters.

The algorithm generalizes to bases bigger than 2, e.g. 10 or 256.

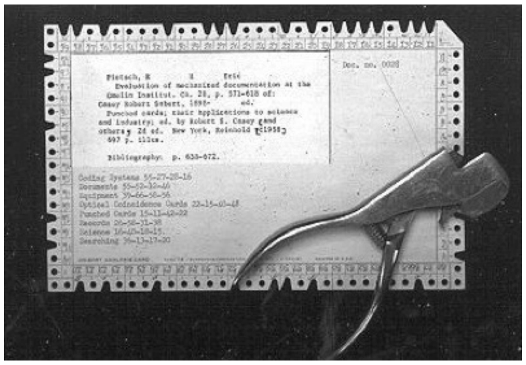
- If the base is 10, split the data into ten groups based on the first digit.
- Then, recursively sort each group starting with the second digit.

Radix sorting (lsb to msb)

A similar method works from least significant bit to the most significant bit.

- First sort on the lsb
- Now sort on the 2ndlsb in such a way that values equal on this bit remain in the same order
- And so on until all bits are processed.

Again the generalization to bigger bases is straightforward.



This can be implemented "by hand" using cards and a skewer.

- One edge of each card is perforated with a pattern of holes and notches.
- A notch is a 1 a hole is a 0
- The spindle is thrust trough the lsb, and all cards with a 0 are picked up and moved to the front.
- Repeat until all bits are processed.

Libraries, businesses, and researchers often used this hand technique at least until the 1980s.

This is again $\Theta(mn)$ (assuming your hand moves with velocity independent of n). However it feels much more like $\Theta(m)$ because much of the work is independent of n, i.e., the time function looks rather like this

$$cm + kmn$$

where c is big and k is small.

A similar method was used for electromechanically sorting punched cards from about 1901 until superceeded by electronic computers.