How long does it take to sort?

We know of several sorting methods (merge-sort, heapsort) that take time $\Theta(n \lg n)$ in the worst case to sort a set of n values.

Can this be improved?

For definiteness, we consider only comparison sorts, which work by comparing pairs of elements. All other logic and moving of elements is ok, but no other use of the elements' values is allowed. The pairs to be compared can be chosen in any way at all.

For comparison sorts, it makes sense to count the number of comparisons.

How long does it take to sort? (continued)

It is easier to analyse a slightly different problem. Consider a list $A = (a_1, a_2, \ldots, a_n)$ of distinct values.

- **Problem 1.** Sort A into increasing order.
- **Problem 2.** Determine which order the elements of A are in. For example the order of $A = (20, 40, 10, 30, 50)$ is $(2, 4, 1, 3, 5)$. There are $n!$ possible answers.

It is easy to see that these two problems require the same number of comparisons: given a solution to one of the problems, it can be modified to solve the other problem without using more comparisons.

How long does it take to sort? (continued)

So suppose we have an algorithm which can always solve Problem 2 in K comparisons.

Each comparison has two possible values "$<$" and "$>$".

Therefore, the sequence of K comparisons has at most 2^K possible values. Such as "$<, >, <, <, >, >, <, <, >, <$".

Since Problem 2 has $n!$ possible answers, it must be that $2^K \geq n!$. Otherwise two different answers would have the same sequence of comparison values.

Conclusion: A comparison sort requires at least $\lg(n!)$ comparisons in the worst case to sort n distinct values.

Finally: $\lg(n!) = \Theta(n \lg n)$.

Chapter 15. Dynamic Programming

Dynamic programming is a method of solving a problem by first solving some subproblems and then combining the results.

This is also true for the **Divide-and-Conquer** method. However, dynamic programming is useful when the subproblems have a large overlap with each other, and maybe even have sub-subproblems in common.

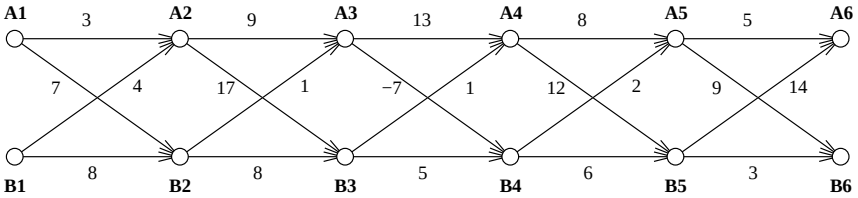
The main requirements for dynamic programming are:

- The total number of (sub-)subproblems which may occur is fairly small.
- The solution to each subproblem can be deduced fairly easily from the solutions to the smaller subproblems.

The basic idea of dynamic programming is to solve the subproblemsfrom smallest to largest, storing the solutions in a table.

Dynamic Programming – multilayer network

Consider a network with n pairs of nodes connected with arrows as in the picture. Each arrow has a length.



The problem is to find the shortest path from the left pair $\{A_1, B_1\}$ to the right pair $\{A_n, B_n\}$.

There are 2^{n-1} such paths, so we don't want to try them all.

Dynamic Programming – multilayer network (continued)

The key observation is:

- Any shortest path from $\{A_1, B_1\}$ to $\{A_n, B_n\}$ consists of a shortest path from $\{A_1, B_1\}$ to $\{A_{n-1}, B_{n-1}\}$ plus one more arrow.

So, let us define some variables:

- $L[X, Y]$ = the length of the arrow from X to Y. (eg. $L[B_2, A_3] = 1$)
- $P(X)$ = the length of the shortest path from $\{A_1, B_1\}$ to X.

Then we have this recurrence:

- $P(A_1) = 0$ and $P(B_1) = 0$
- For $1 \leq i \leq n$:
  $P(A_i) = \min\{P(A_{i-1}) + L[A_{i-1}, A_i], P(B_{i-1}) + L[B_{i-1}, A_i]\}$
  $P(B_i) = \min\{P(A_{i-1}) + L[A_{i-1}, B_i], P(B_{i-1}) + L[B_{i-1}, B_i]\}$

Dynamic Programming – multilayer network (continued)

Very important: What happens if we program this as a recursive function?

The correct method of solution is to apply the recurrence from smallest to largest:

$P(A_1) = 0$; $P(B_1) = 0$
$P(A_2) = 3$; $P(B_2) = 7$
$P(A_3) = 8$; $P(B_3) = 15$
$P(A_4) = 16$; $P(B_4) = 1$
$P(A_5) = 3$; $P(B_5) = 7$
$P(A_6) = 8$; $P(B_6) = 10$.

Therefore, the shortest path from $\{A_1, B_1\}$ to $\{A_6, B_6\}$ has length 8.

* What is the time complexity?
* How do we find the shortest path, rather than just its length?
* What is memoisation?