Sorting algorithms

Input

A list $(a_0, \ldots, a_{n-1})$ of items from a set S with a transitive, reflexive and antisymmetric relation "$\leqslant$" (so that it makes sense to ask someone to sort the list)

Output

A list $(b_0, \ldots, b_{n-1})$ which contains the same elements as the input list and satisfies

$$b_0 \leqslant b_1 \leqslant \cdots \leqslant b_{n-1}.$$

Selection sort

Initialize lists $A = (a_0, \ldots, a_{n-1})$ and $B = ()$
for $k = n-1, n-2, \ldots, 0$ **do**
    Select the largest element a_i of A
    Put $b_k = a_i$
    Remove a_i from A

Example (Initialization)

- $A =$(koala, bilby, possum, echidna, kangaroo, platypus, wallaby, bandicoot)
- $B = ()$

Example

$k = 7$

- A =(koala, bilby, possum, echidna, kangaroo, platypus, ~~wallaby~~, bandicoot)
- B =(wallaby)

$k = 6$

- A =(koala, bilby, ~~possum~~, echidna, kangaroo, platypus, ~~wallaby~~, bandicoot)
- B =(possum, wallaby)

$k = 5$

- A =(koala, bilby, ~~possum~~, echidna, kangaroo, ~~platypus~~, ~~wallaby~~, bandicoot)
- B =(platypus, possum, wallaby)

- etc.

Counting comparisons

How many times do I have to compare two items if I sort a list with $1,000$ items?

A $1,000$

B $999 \times 500 = 499,500$

C $1001 \times 500 = 500,500$

D $1,000,000$

Merge sort

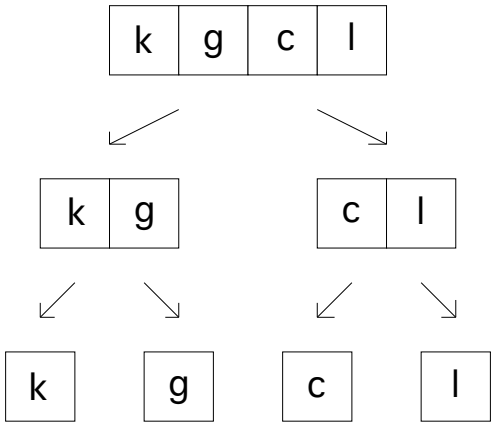
- Assume we have n items to sort
- Put them in an array $A[0], \ldots, A[n-1]$
- do mergesort(A, n), which does
  1. If $n > 1$, do steps 2-7
  2. Set $m = \lfloor \frac{n}{2} \rfloor$
  3. list1 $= A[0], \ldots, A[m-1]$
  4. list2 $= A[m], \ldots, A[n-1]$
  5. mergesort(list1, m)
  6. mergesort(list2, n-m)
  7. A = merge(list1, list2), i.e.
     - 7.1. Look at least item of both lists;
     - 7.2. Choose smaller of the two (or the only choice if one list is empty)
     - 7.3. Put this item at end of output list
     - 7.4. Repeat 7.1 – 7.3 until both lists exhausted

Merge sort

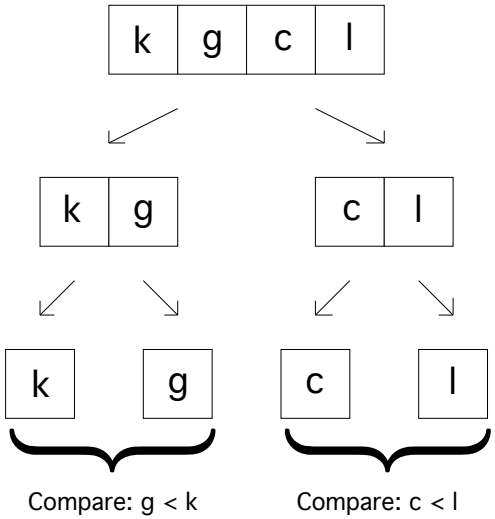
- Merge sort works by breaking the list into halves.
- The halves are then themselves broken into halves, and so on,
- until the sublists have length '1' - which is trivially already sorted.
- The pieces are then successively recombined ('merged')
- making successively longer sublists until we have the whole list

* NB: The method works for any length list (by rounding up or down when needed) but is easiest to analyze for powers of 2, so we will deal with that case only.

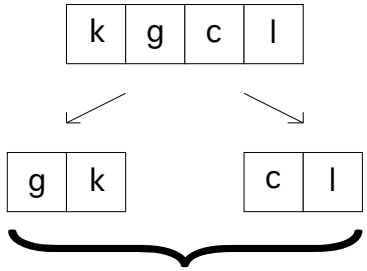
Merge sort: splitting



Merge sort: splitting

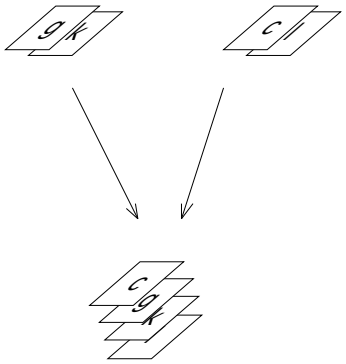


Merge sort: recombining



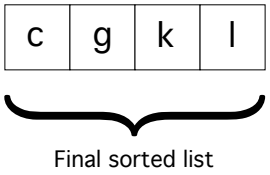
Succesively compare first elements in each sub-list

Merge sort: recombining



- Comparisons:
  - $c < g \implies$ smallest element is c, remaining lists: (g, k) and (l)
  - $g < l \implies$ second element is g, remaining lists: (k) and (l)
  - $k < l \implies$ third element is k and last element is l

Merge sort: final sorted list



Counting comparisons

How many times do I have to compare two items if I sort a list with 4 items?

- A 3
- B 4
- C 5
- D 6

What about 8 items?

- A 10
- B 15
- C 17
- D 24

- For 1,000 items, SelectionSort takes 499,500 comparisons, while MergeSort needs at most 8,977.
- More on sorting: `http://www.sorting-algorithms.com/`

Recursive algorithms

- Merge sort is a recursive algorithm: it calls itself on a smaller problem instance.
- There must always be a base case to prevent infinitely many nested calls.
- This is a very general and powerful principle in algorithm design.
- Example: The Euclidean Algorithm can be written as a recursive algorithm.

Summary

- Sorting is a fundamental algorithmic task that comes up in many applications.
- Selection Sort and Merge Sort are two standard algorithms for this task.
- For a large number of items Merge Sort is the more efficient method.
- A recursive algorithm is an algorithm that calls itself for a smaller problem instance.

Example: Euclidean algorithm

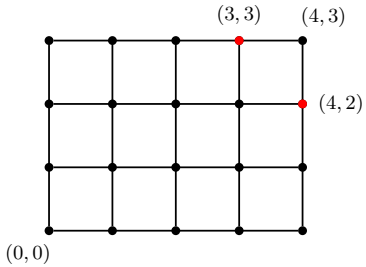
gcd(a, b)

Input: two integers $a \geqslant b \geqslant 0$

if $b = 0$ **then** return a //base case
else
  find integers q and r with
    $a = qb + r$ and $0 \leqslant r \leqslant b - 1$
  return gcd(b, r) //recursive call

Output: the greatest common divisor gcd(a, b)

Another example: counting lattice paths



- What is the number of lattice paths from $(0, 0)$ to $(4, 3)$?
- Every path goes through either $(3, 3)$ or $(4, 2)$.
- Hence the number of paths equals the number of paths to $(3, 3)$ plus the number of paths to $(4, 2)$.

PathNum(m, n)

Input: two positive integers m and n

if $m = 0$ or $n = 0$ **then** return 1 // base case
else
  return PathNum$(m - 1, n)$ + PathNum$(m, n - 1)$ // recursive call

Output: the number of lattice paths from $(0, 0)$ to (m, n)

Merge Sort

`MergeSort`$(a_1, \ldots, a_n)$

Input: a list of items $(a_1, \ldots, a_n)$

if $n = 1$ **then** return (a_1) // base case
else
  Put $m = \lfloor n/2 \rfloor$
  Put $(b_1, \ldots, b_m) =$ `MergeSort`$(a_1, \ldots, a_m)$ // first recursive call
  Put $(c_1, \ldots, c_{n-m}) =$ `MergeSort`$(a_{m+1}, \ldots, a_n)$ // second recursive call
  Put $i = 1$, $j = 1$, and $k = 0$
  while $k < n$ **do**
    increase k by 1
    if $b_i < c_j$ **then**
      Put $a'_k = b_i$ and increase i by 1
    else
      Put $a'_k = c_j$ and increase j by 1
  return $(a'_1, a'_2, \ldots, a'_n)$

Output: the ordered list $(a'_1 \leqslant a'_2 \leqslant \cdots \leqslant a'_n)$