INF421, Lecture 4
Sorting

Leo Liberti

LIX, École Polytechnique, France

Course

- **Objective**: to teach you some data structures and associated algorithms
- **Evaluation**: TP noté en salle info le 16 septembre, Contrôle à la fin. Note: $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$
- **Organization**: fri 26/8, 2/9, 9/9, 16/9, 23/9, 30/9, 7/10, 14/10, 21/10, amphi 1030-12 (Arago), TD 1330-1530, 1545-1745 (SI31,32,33,34)
- **Books**:
  1. Ph. Baptiste & L. Maranget, *Programmation et Algorithmique*, Ecole Polytechnique (Polycopié), 2006
  2. G. Dowek, *Les principes des langages de programmation*, Editions de l'X, 2008
  3. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997
  4. K. Mehlhorn & P. Sanders, *Algorithms and Data Structures*, Springer, 2008
- **Website**: `www.enseignement.polytechnique.fr/informatique/INF421`
- **Contact**: `liberti@lix.polytechnique.fr` (e-mail subject: INF421)

Lecture summary

- Sorting complexity in general
- Mergesort
- Quicksort
- 2-way partition

The minimal knowledge

- `mergeSort`$(s_1,\ldots,s_n)$

  $m = \lfloor \frac{n}{2} \rfloor$;
  $s' =$ `mergeSort`$(s_1,\ldots,s_m)$;
  $s'' =$ `mergeSort`$(s_{m+1},\ldots,s_n)$;
  merge s', s'' such that result $\bar{s}$ is sorted;
  return $\bar{s}$;

  Split in half, recurse on shorter subsequences, then do some work to reassemble them

- `quickSort`$(s_1,\ldots,s_n)$

  $p = s_k$ for some k;
  $s' = (s_i \mid i \neq k \wedge s_i < p)$;
  $s'' = (s_i \mid i \neq k \wedge s_i \geq p)$;
  return (`quickSort`(s'), p, `quickSort`(s''));

  Choose a value p, split s.t. left subseq. has values $< p$, right subseq. has values $\geq p$, recurse on subseq.

- `twoWaySort`$(s_1,\ldots,s_n) \in \{0,1\}^n$

  $i = 1$; $j = n$
  while $i \leq j$ **do**
    if $s_i = 0$ **them** $i \leftarrow i + 1$
      else if $s_j = 1$ **then** $j \leftarrow j - 1$
      else swap s_i, s_j; i++; j-- **endif**
  end while

  Only applies to binary sequences. Move i to leftmost 1 and j to rightmost 0. These are out of place, so swap them; continue until i, j meet

The sorting problem

- Consider the following problem:

  SORTING PROBLEM (SP). Given a sequence $s = (s_1, \ldots, s_n)$, find a permutation $\pi \in S_n$ of n symbols such that: following property:

  $$\forall 1 \leq i < j \leq n \; (s_{\pi(i)} \leq s_{\pi(j)}),$$

  where S_n is the *symmetric group of order* n
- In other words, we want to **order** s

The *type* of s (integers, floats and so on) may be important in order to devise more efficient algorithms: `mergeSort` and `quickSort` are for generic types (we assume no prior knowledge); in `twoWaySort` we know the type is boolean

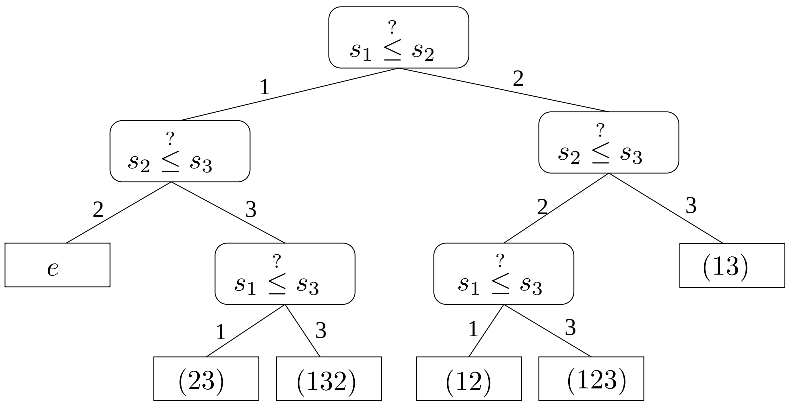
Complexity of a problem?

- Can we ask about the complexity of the sorting problem?
- Recall: usually the complexity measures the CPU time taken by an *algorithm*
- Could ask for the *worst-case* complexity (over all inputs) of the *best* algorithm for solving the problem
- But how does one list *all possible algorithms for a given problem*?

This question seems ill-defined

Comparisons

- The crucial elements of sorting algorithms are **comparisons**: given s_i, s_j, we can establish the truth or falsity of the statement $s_i \leq s_j$
- We can describe *any* sorting algorithm by means of a **sorting tree**
- E.g. to order s_1, s_2, s_3, the sorting tree is as follows:



Sorting trees

- Each sorting tree represents a possible way to chain comparisons as to sort possible inputs
- A sorting tree gives all the possible outputs over all inputs
- Any (comparison-based) sorting algorithm corresponds to a particular sorting tree
- The number of (comparison-based) sorting algorithms is at most the number of sorting trees
- Can use sorting trees to express the idea of *best possible sorting algorithm*

Best worst-case complexity

- Let $\mathbb{T}_n$ be the set of all sorting trees for sequences of length n
- Different inputs lead to different ordering permutations in the *leaf nodes* of each sorting tree
- For a sorting tree $T \in \mathbb{T}_n$ and a $\pi \in S_n$ we denote by $\ell(T, \pi)$ the length of the path in T from the root to the leaf containing π
- Best worst-case complexity is, for each $n \geq 0$:

$$B_n = \min_{T \in \mathbb{T}_n} \max_{\pi \in S_n} \ell(T, \pi).$$

It's remarkable that we can even *formally express* such an apparently ill-defined quantity!

The complexity of sorting

- For any tree T, let $|V(T)|$ be the number of nodes of T
- **Tree depth**: maximum path length from root to leaf in a tree
- A binary tree T with depth bounded by k has $|V(T)| \leq 2^k$
- $\Rightarrow$ The sorting tree T^* of best algorithm has $|V(T^*)| \leq 2^{B_n}$
- $\forall T \in \mathbb{T}_n$, each $\pi \in S_n$ appears in a leaf node of T
- $\Rightarrow$ Any $T \in \mathbb{T}_n$ has at least $n!$ leaf nodes, i.e. $|V(T)| \geq n!$
- Hence, $n! \leq 2^{B_n}$, which implies $B_n \geq \lceil \log n! \rceil$
- By Stirling's approx., $\log n! = n \log n - \frac{1}{\ln 2} n + O(\log n)$
- $\Rightarrow B_n$ is bounded below by a function proportional to $n \log n$ (we say B_n is $\Omega(n \log n)$)

Today's magic result: first part

Complexity of sorting:

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

Simple sorting algorithms

Simple sorting algorithms

- I shall save you the trouble of learning *all* the numerous types of sorting algorithms in existence
- Let me just mention **selection sort**, where you repeatedly select the *minimum* element of s,

$$(3,\boxed{1},4,2) \to (3,4,\boxed{2}),(1) \to (\boxed{3},4),(1,2) \to (\boxed{4}),(1,2,3) \to (1,2,3,4)$$

- and **insertion sort**, where you insert the next element of s in its proper position of the sorted sequence

$$(\boxed{3},1,4,2) \to (\boxed{1},4,2),(3) \to (\boxed{4},2),(1,3) \to (\boxed{2}),(1,3,4) \to (1,2,3,4)$$

- Both are $O(n^2)$; insertion sort is fast for small $|s|$

Mergesort

Divide-and-conquer

- Let $s = (5,3,6,2,1,9,4,3)$
- Split s midway: the first half is $s' = (5,3,6,2)$ and the second is $s'' = (1,9,4,3)$
- Sort s', s'': since $|s'| < |s|$ and $|s''| < |s|$ we can use recursion; base case is when $|s| \le 1$ *(if $|s| \le 1$ then s is already sorted by definition)*
- Get $s' = (2,3,5,6)$ and $s'' = (1,3,4,9)$
- Merge s', s'' into a sorted sequence $\bar{s}$:

$$\begin{matrix}(2,3,5,6)\\(1,3,4,9)\end{matrix} \to (1,2,3,3,4,5,6,9) = \bar{s}$$

- Return $\bar{s}$

Merge

- `merge`(s', s''): merges two sorted sequences s', s'' in a sorted sequence containing all elements in s', s''
- Since s', s'' are both already sorted, merging them so that the output is sorted is efficient
  - Read first (and smallest) elements of s', s'': $O(1)$
  - Compare these two elements: $O(1)$
  - There are $|s|$ elements to process: $O(n)$
- You can implement this using lists: if s' is empty return s'', if s'' is empty return s', and otherwise compare the first elements of both and choose smallest

Recursive algorithm

- mergeSort(s) {
  1: **if** $|s| \leq 1$ **then**
  2: **return** s;
  3: **else**
  4: $m = \lfloor \frac{|s|}{2} \rfloor$;
  5: $s' =$ mergeSort$(e_1, \ldots, e_m)$;
  6: $s'' =$ mergeSort$(e_{m+1}, \ldots, e_n)$;
  7: **return** merge(s', s'');
  8: **end if**

  }

By INF311, `mergeSort` has worst-case complexity $O(n \log n)$

Today's magic result: second part

Complexity of sorting:

$\Theta(n \log n)$

A function is $\Theta(g(n))$ if it is both $O(g(n))$ and $\Omega(g(n))$

Quicksort

Divide-and-conquer

- Let $s = (5, 3, 6, 2, 1, 9, 4, 3)$
- Choose a *pivot* value $p = s_1 = 5$ (no particular reason for choosing s_1)
- Partition $(s_2, \ldots, s_n)$ in s' (elements smaller than p) and s'' (elements greather than or equal to p):

$$(\mathbf{5}, 3, 6, 2, 1, 9, 4, 3) \to (3, 2, 1, 4, 3), (6, 9)$$

- Sort $s' = (3, 2, 1, 4, 3)$ and $s'' = (6, 9)$: since $|s'| < |s|$ and $|s''| < |s|$ we can use recursion; base case $|s| \leq 1$
- Update s to (s', p, s'')

Notice: in `mergeSort`, we recurse *first*, then work on subsequences *afterwards*. In `quickSort`, we work on subsequences *first*, then recurse on them *afterwards*

Partition

- `partition`(s): produces two subsequences s', s'' of $(s_2, \ldots, s_n)$ such that:
  - $s' = (s_i \mid i \neq 1 \land \ s_i < s_1)$
  - $s'' = (s_i \mid i \neq 1 \land \ s_i \geq s_1)$
- Scan s: if $s_i < s_1$ put s_i in s', otherwise put it in s''
- There are $|s| - 1$ elements to process: $O(n)$
- You can implement this using arrays; moreover, if you use a `swap` function such that, given i, j, swaps s_i with s_j in s, you don't even need to create any new temporary array: *you can update s "in place"*

Recursive algorithm

- `quickSort`(s) {

  if $|s| \leq 1$ **then**
    return ;
  else
    $(s', s'') =$ `partition`(s);
    `quickSort`(s');
    `quickSort`(s'');
    $s = (s', s_1, s'')$;
  end if

  }

Complexity

Worst-case complexity: $O(n^2)$

Average-case complexity: $O(n \log n)$

Very fast in practice

Worst-case complexity

- Consider the input $(n, n-1, \ldots, 1)$ with pivot s_1
- Recursion level 1: $p = n$, $s' = (n-1, \ldots, 1)$, $s'' = \varnothing$
- Recursion level 2: $p = n-1$, $s' = (n-2, \ldots, 1)$, $s'' = \varnothing$
- And so on, down to $p = 1$ (base case)
- Each partitioning call takes $O(n)$
- Get $O(n^2)$

2-Way partitioning

Definition by example

Input: $(1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
Desired output: $(0, 0, 0, 0, 0, 1, 1, 1, 1, 1)$

Iterating swaps

- Let $s = (1, 0, 0, 1, 1, 0, 0, 0, 1, 1)$
- Find leftmost 1 and rightmost 0 (these are out of place)
- Swap them
- Increase leftmost counter, decrease rightmost counter
- Repeat until counters become equal

$$(1, 0, 0, 1, 1, 0, 0, 0, 1, 1) \to (\mathbf{0}, 0, 0, 1, 1, 0, 0, \mathbf{1}, 1, 1) \to$$
$$(0, 0, 0, \mathbf{0}, 1, 0, \mathbf{1}, 1, 1, 1) \to (0, 0, 0, 0, \mathbf{0}, \mathbf{1}, 1, 1, 1, 1)$$

- You can implement this using arrays: keep an increasing counter from the first element and a decreasing counter from the last, when out of place swap, end after counters meet

The algorithm

$i = 0$; $j = n - 1$;
while $i \leq j$ **do**
  if $s_i = 0$ **then**
    $i \leftarrow i + 1$;
  else if $s_j = 1$ **then**
    $j \leftarrow j - 1$;
  else
    swap(s, i, j);
    $i \leftarrow i + 1$;
    $j \leftarrow j - 1$;
  end if
end while

Worst-case complexity

- Occurs with input $(1, \ldots, 1, 0, \ldots, 0)$ where number of 1's are around the same as the number of 0's
- Requires $\lfloor \frac{n}{2} \rfloor$ swaps
- Worst-case $O(n)$

A paradox?

- At the outset, we proved that sorting had complexity $\Theta(n \log n)$
- But 2-way partioning requires only $O(n)$
- Contradiction? Paradox?
- Only apparent: the initial theorem was under the following assumptions:
  - no prior knowledge on the type of input *("general input")*
  - only **comparison-based algorithms** are concerned
- Neither assumption is true for 2-way partitioning
  - we know that the input sequence is of binary type
  - the algorithm never uses a comparison

Appendix
[P. Cameron, *Combinatorics*]

Quicksort: average complexity 1/10

- Let $n = |s|$
- Let q_n be the average number of comparisons taken by `quickSort`
- `partition`(s) involves $n-1$ comparisons
- Assume the pivot $p = s_1$ is the k-th smallest element of s
- Then, recursion takes $q_{k-1} + q_{n-k}$ comparisons on average
- Average this over the n values that k can take
- This implies:

$$q_n = n - 1 + \frac{1}{n} \sum_{k=1}^{n} (q_{k-1} + q_{n-k}) \qquad (1)$$

Quicksort: average complexity 2/10

- Notice that in the sum $\sum_{k=1}^{n}(q_{k-1} + q_{n-k})$, each q_k occurs twice

k	q_{k-1}	q_{n-k}
1	q_0	q_{n-1}
2	q_1	q_{n-2}
$\vdots$	$\vdots$	$\vdots$
$n-1$	q_{n-2}	q_1
n	q_{n-1}	q_0

- Hence we can write:

$$q_n = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} q_k \qquad (2)$$

Quicksort: average complexity 3/10

- Equation (2) is a *recurrence relation*
- A *solution* of a recurrence relation is a closed-form expression for q_n which does not include the symbol q_k for any integer $k \geq 0$
- One solution method consists in writing the solution as the infinite sequence $(q_0, q_1, q_2, \ldots, q_n, \ldots)$ as a *formal power series*:

$$Q(t) = \sum_{n \geq 0} q_n t^n \qquad (3)$$

- If $Q(t)$ is known, then the value for each q_n can also be obtained:

  Differentiate $Q(t)$ n times with respect to t, set $t = 0$, and divide the result by n (why does this work?)

Quicksort: average complexity 4/10

- Multiply each side of the recurrence relation (2) by nt^n and sum over all $n \geq 0$, get:

$$\sum_{n \geq 0} nq_n t^n = \sum_{n \geq 0} n(n-1)t^n + 2 \sum_{n \geq 0} \left(\sum_{k=0}^{n-1} q_k \right) t^n \qquad (4)$$

- We now replace each of these three terms so as to be able to derive a more convenient expression for $Q(t)$

Quicksort: average complexity 5/10

- Differentiate $Q(t)$ with respect to t and multiply by t to get an expression for the first term:

$$t\frac{dQ(t)}{dt} = t \sum_{n \geq 0} nq_n t^{n-1} = \sum_{n \geq 0} nq_n t^n, \qquad (5)$$

- We saw in Lecture 1 (proof of Thm. on slide 26) that $\sum_{n \geq 0} t^n = \frac{1}{1-t}$
- Differentiate this equation twice with respect to t, we get:

$$\sum_{n \geq 0} n(n-1)t^{n-2} = \frac{2}{(1-t)^3} \qquad (6)$$

- Now multiply both members by t^2 to get an expression for the second term:

$$\sum_{n \geq 0} n(n-1)t^n = \frac{2t^2}{(1-t)^3} \qquad (7)$$

Quicksort: average complexity 6/10

- Now for the third: the n-th term of the sum $\sum_{n \geq 0} (\sum_{k=0}^{n-1} q_k) t^n$ can be written as

$$\sum_{k=0}^{n-1} t^{n-k}(q_k t^k)$$

- Hence, the whole sum over n can be written as the following product (convince yourself that this is true):

$$(t + t^2 + t^3 + \ldots)(q_0 + q_1 t + q_2 t^2 + q_3 t^3 + \ldots)$$

- The first factor is $\sum_{n \geq 0} t^n = \frac{1}{1-t}$, and the second is simply the expression for $Q(t)$
- Hence, the third term is $\frac{2tQ(t)}{1-t}$

Quicksort: average complexity 7/10

- Putting it all together, we obtain a first-order differential equation for $Q(t)$:

$$tQ'(t) = \frac{2t^2}{(1-t)^3} + \frac{2t}{1-t}Q(t) \qquad (8)$$

- Remark that if we differentiate the expression $(1-t)^2Q(t)$ (*which I pulled out of a hat, or did I?*) w.r.t. t, we get:

$$\frac{d}{dt}((1-t)^2Q(t)) = (1-t)^2Q'(t) - 2(1-t)Q(t) \qquad (9)$$

- We rearrange the terms of Eq. (8) to get:

$$tQ'(t) - \frac{2t}{1-t}Q(t) = \frac{2t^2}{(1-t)^3} \qquad (10)$$

- We multiply Eq. (10) through by $\frac{(1-t)^2}{t}$ and get:

$$(1-t)^2Q'(t) - 2(1-t)Q(t) = \frac{2t}{1-t} \qquad (11)$$

Quicksort: average complexity 8/10

- The RHS of Eq. (9) is the same as the LHS of Eq. (11), hence we can rewrite Eq. 9 as:

$$\frac{d}{dt}((1-t)^2Q(t)) = \frac{2t}{1-t} \qquad (12)$$

- Now, straightforward integration w.r.t. t yields:

$$Q(t) = \frac{-2(t+\log(1-t))}{(1-t)^2} \qquad (13)$$

Quicksort: average complexity 9/10

- The next step consists in writing the power series for $\log$ and $1/(1-t)^2$, rearrange them in a product, and read off the coefficient q_n of the term in t^n. Without going into details, this yields:

$$q_n = 2(n+1)\sum_{k=1}^{n}\frac{1}{k} - 4n \qquad (14)$$

  for all $n \geq 0$

- For all $n \geq 0$, the term $\sum_{k=1}^{n}\frac{1}{k}$ is an approximation of:

$$\int_1^n \frac{1}{x}dx = \log(n) + O(1) \qquad (15)$$

Quicksort: average complexity 10/10

- Finally, we get an asymptotic expression for q_n:

$$\forall n \geq 0 \quad q_n = 2n\log(n) + O(n) \qquad (16)$$

- This shows that the average number of comparisons taken by `quickSort` is $O(n\log n)$