Retrieving Data and Sorting

Suppose we have an array A: A[1],. . . , A[n], such that each element is associated with an *identifier* A[i].id.

Our goal is to find the data associated with a given identifier.

A naive solution can be to scan all the array, and search for our identifier. Asymptotic complexity of this naive solution is O(n), with n size of the array, because maybe we would have to scan all the array to get to our solution (even if we know for sure that the identifier is present in the array, it can be the last element of the array).

Dichotomic search

Suppose that A is sorted wrt the identifiers, namely if $i < j \rightarrow A[i].id < A[j].id$, we are searching for the identifier id_1, start with looking at the identifier associated with the element in the middle of A, i.e. $A[n/2].id$:

- if it is the one we are searching for ($A[n/2].id = id_1$), we are done;
- if it greater than id_1 ($A[n/2].id > id_1$) we have to focus on the first half of the array;
- if it is smaller than id_1 ($A[n/2].id < id_1$) we have to look in the second half of the array;

Repeat as long as A is not empty.

Termination of the dichotomic search: since $r \geq l$ (with r,l right and left extremes indexes of the sub-array of the original array in which we are focused), holds also that $2r \geq r + l \geq 2l$ so that $r \geq \frac{r+l}{2} \geq l$, call $m = \frac{r+l}{2} \in [l, r]$. At the end of an iteration, say that r' and l' are the values of r and l after the iteration, either $r' = m - 1 \leq r - 1$ and $l = l'$, so that $r' - l' \leq r - 1 - l < r - l$, or $r' = r$ and $l' = m + 1$, we and up in a very same situation, so the length of our focus array keep decreasing, so at a certain point the algorithm must terminate. By using more or less the same calculus we end up in computing the complexity of the algorithm.

Complexity of the dichotomic algorithm is $O(log_2(n))$, keep halving the focus array (since the assignment as constant complexity $\Theta(1)$) we end up in a logarithmic time, wrt to the size of the array.

Sorting

Given an array, get the very same array sorted ($i < j \rightarrow A[i] \leq A[j]$). It could be the case that we have multiple instances of the same element inside the array.

Insertion sort

We assume that the first part of the array is already sorted (can also be empty), to increase the number of sorted element we simply need to add the next element (element right after the sorted part of the array) in the right position (inside the sorted array) and we end up in a increased number of elements already sorted. At the very beginning of the process, it is unknown where to place the new element, moreover we don't want to disrupt the already sorted sub-array, so the brutal swap is not feasible.

Instead we can swap an element with the previous one until we end up in the right position (namely until we find the previous one that is greater than it, if any). We keep repeating until the element just before the element we want to insert is $\leq$ of the element we want to insert. Moreover since the element was swapped according to the total order we were considering, we didn't mess up the already sorted sub-array, and so our ordered sub-array is now larger. Keep repeating this process until the end of the array will give a sorted array. If the array is already sorted, the time is linear, as we wanted.

Complexity of the **insertion sort** algorithm is:

- upper bound: $T(|A|) = \sum_{i=2}^{|A|} \sum_{j=1}^{O(i)} \Theta(1) \leq \sum_{i=0}^{|A|} O(i) = O(|A|^2)$, indeed the worst case scenario is that the part of the array is empty (actually consists of only the first element, so the swap actually happens for every element of the array A);

Remark that Θ and big-O are sets, and we are using them in a algebraic way.

- lower bound: $T(|A|) = \sum_{i=2}^{|A|} \sum_{j=1}^{\Omega(i)} \Theta(1) \geq \sum_{i=2}^{|A|} \Omega(1) = \Omega(n)$, indeed the best case scenario is that the array is already sorted and so we actually never swap any element.

In the end, we have a very good performance in the luckier case, almost sorted array, but poor performance in the worst case (better than the naive algorithm, but not the best we can hope for anyway).

Quick sort

Idea is to select a **pivot**, which is able to perform a partition of the array. We simply move elements smaller than the pivot in the first part of the array, elements bigger than the pivot in the final part of the array. Every time we choose a pivot we create a partition of the original array int S, the part of the array containing elements smaller than then the pivot, G, the part of the array containing elements bigger than the pivot, and the pivot itself. We repeat this process on the subarrays having more than one elements, and we are sure that the computation terminates in a sorted array. Indeed at the end of every iteration we get that elements that were in S and G at the previous iteration remain in S and G respectively, while the pivot ends up in its sorted position (notice that S and G are shorted than A by definition).

Hence each iteration places at least one element in the right position and prepares the array for two recursive calls on S and G.

Complexity of the **quick sort** algorithm an be defined by using a recursive equation. The complexity of calling quick sort is $\Theta(1)$ if we are dealing with arrays of length 1, otherwise it depends on the complexity T_P of the partition. Namely it is $T_Q(|A|) = T_Q(|S|) + T_Q(|G|) + T_P(|A|)$.

The choice of the pivot is not relevant in order to guarantee the best performance. However we can select the best way to partition an array.

So, choose a pivot element p:

1. switch the pivot with the first element in A ($\Theta(1)$), for example by swapping (constant complexity $\Theta(1)$);
2. if $A[i] > p = A[1]$, swap $A[i]$ and $A[j]$ and decrease j ($\Theta(1)$);
3. else if $A[i] \leq p = A[1]$, increase i;

At each iteration we are decreasing the distance between i and j, so the number of step is the distance between j and i at the beginning of the loop, so number of repetitions is $\Theta(r - l)$, each of constant complexity $\Theta(1)$

4. Repeat until $i \leq j$ (until there is some something to be processed) and swap $p = A[1]$ and $A[j]$ (meaning as many times as the number of elements of A, asymptotically), this takes $\Theta(1)$ (during the iterations, i is the index of the last element of S, j is the index of the first element of G. After all iterations, i is the index of the pivot, which is now placed between S and G).

The complexity of this partition procedure is $T_P(|A|) = \Theta(|A|)$.

The complexity of performing quick sort on an array of length |A| is: $T_Q(|A|) = T_Q(|S|) + T_Q(|A| - |S| - 1) + \Theta(|A|)$ (partition and quick sort on both sides of the partition).

In the worst case scenario |S| = 0 (or |G| = 0, already sorted array, since we would select the smallest element in the array as pivot, since our choice is to take the leftmost element as pivot; this would be different if for example we chose to select as pivot the element in the middle, in that case the already sorted array would be the best case scenario); the above equation becomes $T_Q(|A|) = T_Q(0)+T_Q(|A|-1)+\Theta(|A|) = \Theta(1)+T_Q(|A|-1)+\Theta(|A|)$, unravelling the sum we have $T_q(|A| - 1) = O(\sum_{i=0}^{|A|} i) = O(\frac{|A|(|A|+1)}{2}) = O(|A|^2)$ (the point is that, no matter how we select the pivot, there will always be a worst case scenario in which the complexity is quadratic).

So quick sort, in the worst case, as the same complexity as insertion sort.

Best case scenario, we are guaranteed at each step to partition the array in two subarrays such that the length of $|S| = \frac{1}{10}|A|$ and $|G| = \frac{9}{10}|A|$ (balanced partition), in this case we have that the height of the leftmost branch (dividing n

by 10 at each step) of the recursion tree is $\log_{10}(n)$ (number of steps required to go from n to 1 diving by 10 each step), while in the rightmost branch (divide by $\frac{10}{9}$) the height of the recursion tree is $\log_{\frac{10}{9}}(n)$. Partition procedure at level one takes time $c \cdot n$, the left part will end up in having $\frac{1}{10}n$ elements, the right part $\frac{9}{10}n$, at each level this factors propagate, so the cost of executing the partition is linear ($\Theta(n)$ represented by $c \cdot n$), until the last level of the leftmost branch, so the lower bound for the quick sort is $\Omega(n \cdot \log(n))$. Moving down, the number of steps is $\leq$ than $c \cdot n$, since the length of the rightmost branch of the tree is logarithmic wrt n, so the upper bound is $O(n \cdot \log(n))$, hence the overall complexity is $\Theta(n \log(n))$ (in the case of this kind of partition, whenever we can specify a fixed ratio between the length of S and the length of G we can repeat this reasoning). Average case: all permutations of A are equally likely, so the partition has a ratio more balanced than $\frac{1}{d}$ with probability $\frac{d-1}{d+1}$ Average case if good and bad cases alternate, we end up with a complexity of $\Theta(n \log(n))$.

So insertion sort outperforms quick sort in the best case scenario (already sorted array), however it is really rare to get the best case scenario, so most of the time we end up in quadratic complexity. With quick sort is quite easy to have a "good" case scenario, so most of the time we end up better than in insertion sort.

Summary of quick sort: it applies the divide-and-conquer strategy, in three steps:

- *divide*: partition the array into two (possibly empty) subarray, in such a way that all the elements in S are smaller than the pivot, which is itself smaller than every element in the array G;
- *conquer*: sort the two subarrays S and G by recursive calls to quick sort;
- *combine*: since the subarrays are already sorted, no work is needed to combine them, hence at the end the whole array is sorted.

The key of the algorithm is the partition procedure, which arranges the subarrays in place.

The running time of quick sort depends on whether the partition is balanced or unbalanced.

Finding the maximum

A possible way to sort an array is finding the maximum and place it at the end of the array (last position). Repeat in the initial fragment of the array, until we completely sort all the array. Complexity of performing all these operations is $\sum_{i=1}^{|A|}(T_{max}(i) + \Theta(1))$.

Now we need to define a way for finding the maximum. **Bubble sort** consists in pair-wise swapping the maximum to the right. We start from the beginning of the array and we compare each element with its successors (in the initial, unsorted part of the array), until we find an element bigger than our current element. In detail:

- scan the array A, i (in |A|...1) is the position in which I want to insert the maximum, j (in 1...i-1) is the element under examination, the number of repetition of the inner loop is $\Theta(i)$.
- if A[j] > A[j+1], swap A[j] and A[j+1] (at this step the maximum is j+1), this step has complexity $\Theta(1)$.

The complexity of this algorithm is $T_B(|A|) = \sum_{i=1}^{|A|} \sum_{j=1}^{i-1} \Theta(1) = \Theta(|A|^2)$.

Hence we can deduce that bubble sort is a quite inefficient sorting algorithm.

Another way of finding the maximum could be to linear scan the unsorted part of the array: **selection sort**. The selection sort proceeds as:

- scan all the unsorted part of the array and compare the element pair-wise, tracing the max among them;
- swap the max just found with element before the start of the sorted part of the array.

The complexity of selection sort among an array of length |A| is $T_S(|A|) = \sum_{i=1}^{|A|} \sum_{j=2}^{i} (\Theta(1)) + \Theta(1) = \sum_{i=1}^{|A|} (\Theta(i) + \Theta(1)) = \Theta(|A|^2)$.

We want to find the maximum element in the unsorted part of the array with a complexity less than $\Theta(|A|^2)$.

Heap sort

In order to find the maximum we can use **max-heap**: we store the elements of A in a max-heap H_{max} (so that now A is empty), we extract the min (i.e. the max in max-heaps) and place it in A (start to empty A and repopulate A), repeat until the heap is empty.

On an array based representation of the heap, we can do this all in place.

Complexity is:

- Building the heap costs $\Theta(n)$;
- Extract the min costs $O(\log(i))$ where i is the position in which we want to place the element (i-th iteration, storing in i-th position) at iteration;

Hence we have $T_H(n) = \Theta(n) + \sum_{i=2}^{n} O(\log(i)) \leq O(n) + O(\sum_{i=2}^{n} \log(n)) = O(n \log(n))$. This outperforms all the algorithm we saw before.

Hence the heapsort algorithm starts by using *build_heap*, to build a max heap on the input array A. Since the maximum element is stored at the root, we can put it into its correct final position by exchanging it with $A[n]$. If we now discard node n from the heap, we see that the children of the root remain max-heap, but the new root might violate the max-heap property, hence we would to call *heapify*, which leaves a max heap in $A[1, ..., n-1]$. the heapsort algorithms then repeats this process for the max heap of size *n-1* down to heap of size 2.

Summarising: heap sort is a comparison based algorithm, it can be thought as an improved selection sort: like selection sort, heapsort divides the input in a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region;; rather, heapsort maintains the unsorted region in a heap data structure to more quickly find the largest element in each step. It is an in-place algorithm, but it is not a stable sort (stable sort algprithms sort repeated elements in the same order that they appear in the input, when equal elements are indistinguishable, stability is not an issue).

Sorting by comparison

Our goal now is to find an algorithm able to sort an array in linear time.

We want to summary the execution of a sorting-by-comparison algorithm in a decision tree. We end up in having on the leaves all the permutations of elements of the array A, namely there are n! leaves (any comparison between two elements corresponds to a node which branches the computation according to whether $\leq$ or $>$).

The execution of the sorting algorithm corresponds to tracing a single path from the root of the decision tree, down to a leaf. The length of the longest path simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of a any comparison sort algorithm.

The height is maximum number of comparisons required by the algorithm, since a binary tree has no more than 2^h leaves, we get $h \geq \log_2(n!) \in \Omega(n \log(n))$, so the heap sort asymptotically is the best we can get.

No possible to find a *general* algorithm to solve a sorting problem in linear time.

However this is possible for specific cases, for example if we know that elements of A are all in [1,k] (bounded domain for the array values), with k constant (and uniform distribution of the array values), with an algorithm called **counting sort**. Indeed we can allocate another array of k elements, say C, and the i-th element represents the number of elements is A smaller than i+1 (this is done by scanning A from left to right). After scanning A, we have that C, for each index i, contains the number of occurrences in A of the element i. Now we sum up from the left to the right all the elements of C (prefix sum in C).

We now need a third array B, having same length as A. In order to fill B we scan A from right to left and we find in C the correspondent cell (cell having as

index the key in A) and we place the value of A in B in the cell indexed by the value contained in C, and we decrease the value in C.

We move from right to left, because in this way we guarantee that the relative order of equal values is preserved (*stability*).

This algorithm is easily generalisable to generic intervals $[k_1, k_2]$. The cost of this algorithm is:

- initialisation of C takes $\Theta(k)$;
- scanning A and inserting in C takes $\Theta(|A|)$;
- summing in C takes $\Theta(k)$ (repeating k-1 times an assignment and a sum, which takes constant time);
- going from A to B has complexity again $\Theta(|A|)$.

Hence the overall complexity of counting sort is $T_C(n, k) = \Theta(k) + \Theta(|A|) + \Theta(k) + \Theta(|A|) = \Theta(k + |A|)$.

Notice that we are not comparing any value, and we are assuming that all values are included in the interval [1,k], which is a parameter of the complexity, because increasing k the complexity increase. Of course this algorithm is not in place.

Counting sort beats the lower bound $\Omega(n \log(n))$, because it is not a comparison sort. Instead counting sort uses the actual values of the elements to index into an array.

Radix sort

The idea to sort an array A of d-digit values, digit by digit. Counterintuitively we sort the digits starting from the *least significant digit*.
We proceed as:

- for each digit i from the rightmost to the leftmost (looking exclusively one digit at a time);
- to guarantee the correct behaviour of radix sort we need to use a stable algorithm to sort the digits (for example counting sort algorithm is perfect to deal with digits).

In this way we are able to sort values which are far away from each other, avoiding parameters such as in counting sort.

Complexity of **radix sort**: need to take into account the complexity of digit sorting, for example for counting sort it is $\Theta(k + |A|)$, then radixx sort has complexity $\Theta(d(|A| + k))$, with d = number of digits in each of A's values.

Bucket sort

We assume a uniform distribution of the values in A wrt the interval [0,1), we proceed as follows:

- we split [0,1) in n buckets (n=|A|): $[\frac{i-1}{n}, \frac{i}{n})$, for $\forall i \in [i, n]$ and we allocate an array B having length n;
- we add each value of A to the correct bucket (buckets are sets, arrays!), and we end up with the array of buckets;
- sort the buckets (with any sorting algorithm), since we are assuming uniform distribution in [0,1), we will have in average one element per bucket, so we can sort them in constant time (remember that each value is an array);
- scan the array B and for each bucket reassign the value in A (reverse content of the buckets in bucket order on A);

Overall (expected) complexity is $O(n)$ (this is the average running time), indeed:

- $\Theta(n)$ initialize B;
- $\Theta(n)$ filling the buckets;
- $O(n)$ sorting the bucket, assuming uniform distribution for values in [0,1) (this is the expected complexity);
- $O(n)$ reverse buckets' content into A.

Like counting sort, bucket sort is fast because it assumes something on the input.

Select

Consider an unsorted array A, we want to find the element that would be in position 1 after sorting A: this would take $\Theta(n)$, since we have to scan all the array. If we want to find the element in position n after sorting A, again we have complexity $\Theta(n)$. If we want to determine the minimum and the maximum of a set of n elements, we can easily obtain an upper bound of $n - 1$ comparisons (examine each element of the set in turn and keep track of the smallest/larger seen so far).

If instead we want to find the element in general position $i \in [1, n]$ after sorting A, than complexity would be $O(n \log(n))$.

Consider the **select** problem:

- INPUT: a potentially unsorted array A and an index $i \in [1, |A|]$;
- OUTPUT: the value $\bar{A}[i]$, with $\bar{A}$ sorted version of A (namely we want to find the element in A which is larger than exactly $i - 1$ other elements in A).

However this problem can be reformulate, so that the output is an index j for $\tilde{A}$ such that it is exactly $\bar{A}[i]$.

We don't want to build an index, the problem is going to be solved once and for all (una-tantum query).

To simplify the problem, at the beginning we assume that A does contain multiple instances of same value.

Idea is to select a pivot in the array A (not important how we select the pivot), we compute a partition (getting k, position of the pivot after partition procedure) and get two sub-array S (smaller than the pivot) and G (bigger than the pivot). After that we can compute the index i (which is one of the inputs of our problem), and compare i and k: if i=k, then return A[k], if i<k then $\bar{A}[i]$ is in S, in this case we reiterate the search focusing on S, if i>k then $\bar{A}[i]$ in in G, so we focus and repeat the search on G (this is a *dichotomic approach*, but with unsorted array). Once we apply the partition, we focus only in one sub-array. We are substantially describing a recursive algorithm to solve the problem.

As in quick sort, the choice of the pivot makes the difference (in quick sort, the unbalanced worst case scenario was $\Theta(n^2)$, the best case would be to choose the median of the sorted array). Our strategy in now to try to guess an almost median (nearby median) value for $\bar{A}$.

Possible way to select a pivot for improving both the complexity of quick sort and select:

- split A in $\lceil \frac{n}{5} \rceil$ chunks (each of size 5, except for the last one which is $n\_mod(5)$);
- find the median m_i of C_i (for example by sorting C_i, in constant time for each chunk, since the size is fixed), so this step takes time $\Theta(n)$, with n size of the original array;
- once we have the median for each chunk, we can recursively (calling select) compute the median of the medians (again split in chunks the array made by the medians and go on, until the array is bigger than a given size);
- partition the input array around the median of medians (using a modified version of *partition* used for quick sort).

Note that if n is odd, the median is unique, otherwise there are two of them. Regardless of the parity of n, medians occur at $i = \lceil \frac{n+1}{2} \rceil$ (*upper median*) and $i = \lfloor \frac{n+1}{2} \rfloor$ (*lower median*).

We now want to check whether selecting as pivot the median of the medians has partitioned A in a even way.

Consider each chunk as column of a matrix (this is another way to see the whole array), then we can select:

- how many of them contains the median which is $\geq$ the pivot we have selected (namely m, the median of the medians). By definition of median, they are $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil$;
- then among them we search for the number of chunks that have at least 3 elements greater than the pivot (this cannot be done if the array has repetitions), they are all except for the one group that has fewer elements

than 5 (if n is not exactly divided by 5) and the chunk containing m itself (by definition of median), hence $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2$;

- thus the least number of elements greater than the median $3 \cdot \left(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2 \right) \geq \frac{3n}{10} - 6$.

Similarly we have the same lower bound for the number of elements less than the median of medians.

Complexity of select algorithm: $T_s(n) = T_s(\lceil \frac{n}{5} \rceil) + T_s(\frac{7}{10}n + 6) + \Theta(n) =$ compute the median of medians(medians of chunks) + recursive step.

We can solve this recursive equation by *substitution method* (compute complexity by induction). Need to guess $T_s(n)$ complexity: let us guess $T_s(n) \in O(n)$, select a representative of O(n), we also need a representative for the term $\Theta(n)$ in $T_s(n)$.

Assume that $T_s(m) \leq cm \ \forall m < n$ (this is a reference for the definition of big-O), so let us prove that $T_s(n) \leq cn$ (we want to prove that this function is in big-O, if we wanted to prove that it is in $\Theta(n)$, we have to reverse this relation).

Since we know that $T_s(n) = T_s(\lceil \frac{n}{5} \rceil) + T_s(\frac{7}{10}n + 6) + c'n$
$\leq (if \ n \geq 5, \ and \ \frac{7}{10}n + 6 < n \ since \ n > 0) c \cdot \lceil \frac{n}{5} \rceil + c(\frac{7}{10}n + 6) + c'n$
$\leq (\frac{n}{5} + 1) + c(\frac{7}{10}n + 6) + c'n$
$\leq c(\frac{9}{10}n) + c7 + c'n$

We have that $\frac{7}{10}n + 6 < n \leftrightarrow n > 20$, moreover if $c \geq c'$, then $T_s(n) \leq \frac{9}{10}cn + \frac{1}{20}cn \leq \frac{9}{10}cn + c7 \leq cn$ when $n \geq 140$.

If we take $c \geq 20c'$, with c' representative of sorting the whole chunks, and for $n > 140$, then we are able to prove $T_s(n) \in O(n)$ by induction.

We can use this machinery of the median of medians also as pivot in quicksort, yielding an optimal worst case complexity of $O(n \log(n))$.