Modular Smoothed Analysis[1]

M. Schellekens[2], Aoife Hennessy[3], Bichen Shi[4]

Abstract

Spielman's smoothed complexity - a hybrid between worst and average case complexity measures - relies on perturbations of input instances to determine where average-case behavior turns to worst-case. The paper proposes a method supporting modular smoothed analysis. The method, involving a novel permutation model, is developed for the discrete case, focusing on randomness preserving algorithms.

This approach simplifies the smoothed analysis and achieves greater precession in the expression of the smoothed complexity, where a recurrence equation is obtained as opposed to bounds. Moreover, the approach addresses, in this context, the formation of input instances–an open problem in smoothed complexity.

To illustrate the method, we determine the modular smoothed complexity of Quicksort.

Keywords: Smoothed analysis, $\mathcal{MOQA}$, recursive partial permutation model

[1]This grant was supported by SFI grant SFI 07/IN.1/I977. The authors are grateful for fruitful discussions with D. Early that have improved the presentation of the paper and support by Ang Gao in preparing the diagrams.

[2]Centre for Efficiency Oriented Languages(CEOL), University College Cork, Ireland, m.schellekens@cs.ucc.ie

[3]Centre for Efficiency Oriented Languages(CEOL), University College Cork, Ireland, a.hennessy@cs.ucc.ie

[4]Centre for Efficiency Oriented Languages(CEOL), University College Cork, Ireland, b.shi@umail.ucc.ie

1. Introduction

Smoothed Analysis is a framework for analyzing algorithms and heuristics, partially motivated by the observation that input parameters in practice often are subject to a small degree of random noise [17]. In smoothed analysis, one assumes that an input to an algorithm is subject to a slight random perturbation. The *smoothed measure* of an algorithm on an input instance is its expected performance over the perturbations of that instance. The *smoothed complexity* of an algorithm is the maximum smoothed measure over its inputs. The area has been widely studied, following up on the ground breaking work by Spielman and Teng on the smoothed analysis of the simplex method [16]. For an overview of the literature, we refer the reader to the survey paper [17].

In this paper, we focus on exploring a modular approach to smoothed analysis. Modularity is a property of systems (hardware or software), which reflects the extent to which it is decomposable into parts, from the properties of which one is able to predict the properties of the whole[12]. Modularity brings a strong advantage. The capacity to combine parts of code, where the complexity is simply the sum of the complexities of the parts, is a very helpful advantage in static analysis.

We will focus on exploring the question on whether a modular approach can be achieved for smoothed analysis in the discrete context, i.e. focus on a technique that allows one to extract the smoothed complexity of an algorithm from the smoothed complexity of its components. We propose a new modular technique and illustrate the approach on the Quicksort Algorithm. The method yields a recurrence equation expressing the exact (modular) smoothed complexity $T_{QS}^{MS}(n,k)$ of the algorithms under consideration. We obtain the following recurrence[5] for the modular smoothed complexity $T_{QS}^{MS}(n,k)$ of Quicksort:

The modular smoothed complexity of Quicksort, $f(n,k)$, is obtained (Eq. 14) as,

$$f(n,k) = (n-1) + \sum_{j=1}^{n} \beta_{n+1-j}^{n} f(j-1,k) + \sum_{j=1}^{n} \beta_{j}^{n} f(j-1,k), \quad (1)$$

[5]Where, as discussed in the paper, we work with a perturbation probability σ approximated by k/n, for a partial permutation of k elements out of n, in order to express the smoothed complexity as a recurrence.

where

$$\beta_n^n = \frac{n-k+1}{n},$$

and

$$\beta_i^n[i \neq n] = \frac{(k-1)}{n(n-1)}.$$

Since the argument of [4] cannot rely on subproblems generated in recursive calls being again random permutations (for the case of partial permutations), the authors presented an alternative argument based on randomized incremental constructions [11]. This problem is overcome with our modular approach and allows us to form a recursive argument.

The authors of [4] state in "Pitfalls" that "the expected running time of quicksort on random permutations can be analyzed in many different ways. Many of them rely on the fact that the subproblems generated by recursive calls are again random permutations. This is not true for partial permutations ..."

This is consistent with our experience. As discussed in [18], algorithms that do not preserve randomness are typically hard or impossible to analyze. A case in point is Heapsort, for which the exact average-case time is unknown to date for all Heapsort variants [6]. Heapsort's average time has been obtained by the incompressibility method [10], without the constants being determined for the linear factors. [18, 7] presents an alternative algorithm, Percolating Heapsort (taking fewer comparisons than Heapsort), for which the average-case analysis is straightforward and the constants involved have been determined, including for the linear terms. The analysis of Percolating Heapsort only takes a few lines, which may be contrasted with the argument by contradiction presented in [15] and the Kolmogorov complexity argument of [10]. The difference arises from the fact that Percolating Heapsort is randomness preserving, while, as observed in [9], the selection phase of Heapsort is not.

The situation with the smoothed analysis of Quicksort is similar. Because there is no guarantee that subproblems generated by recursive calls are again random permutations, for the case of partial permutations, the analysis is hard and needs to rely on a case based analysis involving randomized incremental constructions. To determine a precise asymptotic constant for one case, the sorted list, requires an involved argument, as pointed out by the authors in a final comment after the proof:"When we consider partial permutations of a sorted sequence (the worst-case instance without permutations)

we are able to get closed form formulae for the $X'_{ij}s$. We distinguish 10 subcases, most of them involving 7 nested sums. From these sums (involving binomials), it is possible to get the differential equation satisfied by their generating functions, and then the Frobenius method allows to get the full asymptotic scale which gives a $2pnln(n)$ complexity. We refer the reader to the full paper for details."

Our approach, in contrast, involves subproblems generated by recursive calls which are again random for the case of partial permutations. As a result we can carry out a more fine tuned analysis, relying on very basic combinatorial arguments.

The model we propose to compute the smoothed complexity is new. As pointed out by Spielman and Teng in the survey paper "Smoothed Analysis of Algorithms and Heuristics: Progress and Open Questions" [17]: "one must understand the possible limitations of any particular permutation model, however, and not overstate the practical implication of any particular analysis. One way to improve the similarity-or-distance based model is to develop an analysis framework that takes into account the formation of input instances. For example, if the input instances to an algorithm A come from the output of another algorithm B, together with a model of B's input instances, then algorithm B together with a model of B's input instances, is the description of A's inputs." We will refer to the algorithm B in the following as the "feeder" algorithm" for A.

The above approach is precisely the model we propose in the current paper in the context of discrete randomness preserving algorithms. Many sorting and search algorithms are randomness preserving, including Quicksort, its median-of-three variant, Insertionsort, Mergesort, Quickselect, Treapsort and a variant of Heapsort, referred to as Percolation Heapsort ([18, 27]). For such randomness preserving algorithms, the feeder algorithm B's output instances (the inputs for A) can be nicely captured via the notion of a random bag [18, 22]. We will use this to formulate a novel model for smoothed complexity, to achieve modular smoothed analysis.

Our approach relies on tracking the data throughout the computation, where the notion of property preservation, as discussed in [17] plays a role. Following [17]:"as a purpose of smoothed analysis is to shed light on practical problems it is more desirable to use a perturbation model that better fits the input instances. Thus, if all or most practical instances of an algorithm share some common structures, such as being symmetric or being planar, then to have a meaningful theory, one may have to consider perturbations

that preserve these structures ... So far, however, the smoothed complexities of various problems and algorithms under these perturbations remains wide open ... Given a property P and a basic model of perturbation, a P-preserving perturbation of an object $\overline{X}$ is a perturbation X of $\overline{X}$ according to the basic perturbation model, but subject to the condition $P(X) = P(\overline{X})$. In the case when $\overline{G}$ is a graph the basic model e.g. is the σ-perturbation of $\overline{G}$[6] The better we can model our input data, the more accurately we can model the performance of an algorithm." [17].

In this spirit we will focus on modelling data throughout the computation, where the property of "random bag representation", a fundamental property to ensure a modular approach [18, 22], will be preserved. Many sorting and search algorithms are random bag preserving [18] and hence allow for a natural representation of the data via random bags and the tracking of these random bags throughout the computation. We will illustrate this approach for the Quicksort algorithm below.

For these algorithms, we will illustrate that a random bag representation of their input data is preserved under perturbations (partial permutations). For this purpose, our analysis starts with the perturbation of input partitions (as opposed to single inputs, regarded in our context as a partition of singletons).

The input partitions and their perturbations, will be tracked throughout the computation, where, at each stage, the newly produced outputs will be perturbed yet again, and passed on to the next basic operation involved in the algorithm. For Quicksort-type algorithms, the approach focuses on tracking input partitions and their perturbations under applications of the basic split operation involved in the recursion. That is, our natural model of perturbation is Partial Permutations that are applied throughout the computation.

Smoothed analysis, based on this approach, yields a traditional recurrence equation.

In our model, the data will be represented as finite labelled partial orders, or LPOs, and random structures.

A *labelled partial order*, or LPO, is a triple $(A, \sqsubseteq, l)$, where A is a set, $\sqsubseteq$ is a partial order on A (that is, a binary relation which is reflexive, anti-

[6]Insert every non-edge into the graph with some probability σ and delete every edge with the same probability.

symmetric, and transitive), and the labeling l is a bijection from A to some totally ordered set C which is increasing with respect to the order $\sqsubseteq$.

Given a finite partial order $(A, \sqsubseteq)$ and a totally ordered set C with $|C| = |A|$, the *random structure* $R_C(A, \sqsubseteq)$ is the set of all LPOs $(A, \sqsubseteq, l)$ with $l(A) = C$.

As the algorithms we consider are comparison-based, the choice of the set C is unimportant, and we will generally write a random structure as $R(A, \sqsubseteq)$, without the subscript[7].

The algorithms we consider have the property of "randomness preservation", which means that applying any operation to each LPO in a random structure results in an output isomorphic to one or more random structures, which is the key to systematic timing.

Formally: a *random bag* is a multiset[8] of random structures. We represent random bags using the multiplicity notation for multisets, so that, for each i, the random bag $\{(R_i, K_i)\}_{i=1,\ldots,n}$ contains K_i copies of the random structure R_i. A function which takes a random structure as argument is random bag preserving if its output is a random bag $\{(R_i, K_i)\}_{i=1,\ldots,n}$.

In this context, any sorting algorithm transforms the random structure with underlying discrete order of size n, i.e. a random bag containing a single random structure R_1 with multiplicity one, into the random bag containing a single random structure R_2 with underlying linear order with multiplicity $n!$. Here, R_1 comprises of exactly $n!$ labelings, representing the traditional $n!$ input lists used in the analysis of sorting algorithms. The random structure R_2 has exactly one labeling, representing the sorted list, of which $n!$ copies will be produced by any sorting algorithm.

As computations proceed in our model, the partial orders underlying the random structures will become more refined (i.e. more order is introduced)[9]. For instance, any sorting algorithm will start its computation from a random structure with underlying order the discrete partial order. It will transform this into the sorted output, hence into the random structure with underlying

[7]More formally, we can consider the random structure to be the quotient of the set of all LPOs on the partial order $(A, \sqsubseteq)$ with respect to a natural isomorphism. See [18, 22] for a full discussion.

[8]I.e. a set which allows elements to occur with any integer multiplicity, rather than only zero or one.

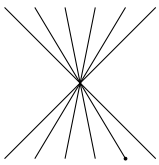
[9]Formally: A poset $(X_1, \sqsubseteq_1)$ is a refinement of another poset $(X_2, \sqsubseteq_2)$ if for all $x, y \in X_1. x \sqsubseteq_2 y \Rightarrow x \sqsubseteq_1 y$.

partial order the linear partial order, a refinement of the discrete partial order.

To set the stage for the analysis of Quicksort, we briefly discuss the split operation, recursively called by Quicksort to order the data. In the traditional version of Quicksort, a fixed pivot is chosen. Without loss of generality, we assume this pivot is the first element in the list. The split operation reorders a given list, placing all elements smaller than the pivot before the pivot (in the same order as encountered in the original list) and all elements greater than the pivot after the pivot (again in the same order as encountered in the original list). The usual representation of the output list is a list in which the pivot element sits in its "correct" position. I.e. if the pivot has relative rank i among the elements in the list, the pivot will be placed in the i-th position in the output list, all elements smaller than the pivot will occur to its left and all pivots greater than the pivot will occur to its right.

The underlying partial order for which this output list is a labeling is the "star-shaped order" containing a central element (to store the pivot label), with $n - i - 1$ elements placed above it (to store the labels greater than the pivot) and $n - i$ elements below it (to store the labels less than the pivot).

$n - i - 1$ elements



i elements

Split is a random bag preserving operation. Here, we illustrate this fact on lists of size 3. In a nutshell, the random bag presentation of the outputs of split corresponds to representing the "order-information" gathered during the execution of the split operation. The elements placed by split to the left of the pivot, will be represented as "smaller" than the pivot, i.e. below the pivot as labels on the above Hasse diagram. The elements placed by split to the right of the pivot, will be represented as greater than the pivot, i.e. above the pivot as labels on the Hasse diagram. We illustrate this with the image below, where for each single list, the resulting labeling is displayed.

In figure 1, it is clear that the split operation maps the random structure over the discrete partial order of size 3 (six labelings corresponding to the 3!

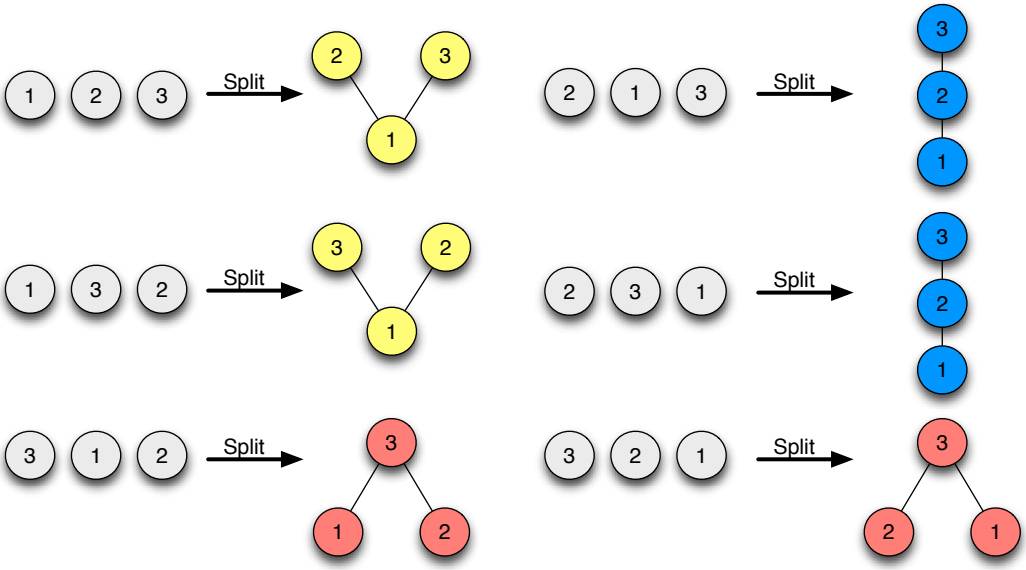


Figure 1: MOQA split: $R(...) \to (R(\vee), 1), (R(|), 2), R(\wedge, 1)$.

input lists of size 3) to the random bag consisting of three random structures. The random structure over the $\vee$ shaped partial order, the random structure over the $\wedge$ shaped partial order and two copies of the random structure over the linear order of size 3.

The model proposed here can be viewed as a new step to explore the discrete case: the points in our space are the random structures (or more generally random bags). The perturbations happening throughout the computation, and acting on random structures, will become proportionally less in relation to the amount that our data is already sorted. The sorting computation gradually "refines" (= introduce more order on) the random bags after each operation. The perturbations will never affect already sorted elements (in this case the pivots placed in their correct position after each split) and ultimately, in the last step of the recursion, will not affect the sorted output. Formally, our perturbations on a given LPO can only act on "free pairs" of labels, i.e. labels that, when swapped result in a new LPO. For this to occur, the swapped labels need to respect the underlying partial order.

In this paper, we define the smoothed complexity of a partition $P, P =$

$\{P_1, \ldots, P_m\}$ of $\sum_n$ as

$$T_A^S(P,k) = max_{i\in\{1,\ldots,m\}}\overline{T}_A(Pert_{k,n}(P_i)),$$

where $Pert_{k,n}(P_i)$ is the set of outputs from permutations of P_i after perturbation of k element of the permutation of size n. We take the maximum over the set of partitions of $\sum_n$, $\mathcal{P} = Partitions(\sum_n)$, where $P \in \mathcal{P}$ is denoted by $P = \{P_1, \ldots P_m\}$ and $i \leq m \leq n!$ and refer to this approach of incorporating all possible input samples in a smoothed complexity analysis as *sample-fairness*.

We define the sample-fair smoothed measure as follows:

$$T_A^{SFS}(n,k) = max_{P\in\mathcal{P},P=\{P_1,\ldots,P_m\}}[T_A^S(P,k)] \tag{2}$$

It is clear that the smoothed complexity is the restriction of the fair smoothed complexity to a single partition P of $\sum_n$ consisting of all singletons, i.e. $P = \{\{s_1\}, \ldots, \{s_{n!}\}\}$, which we refer to as the *singleton base* in the following. Hence $T_A^{SFS}(n,k) \geq T_A^S(n,k)$.

Even though the fair smoothed complexity seems to take into account more sample spaces and hence may yield a different result, we will show that this is not the case.

Theorem 1. $T_A^{SFS}(n,k) = T_A^S(n,k)$.

Theorem 4 allows us to develop our new model, where we compute the smoothed complexity relying on a different base than the singleton base. For this purpose we will select a new base, referred to as a *modular base*. Any random bag preserving operation has a modular base, consisting of the partition of its input data that consists of inverse images of the random structures in its output random bag. For the split operation of Quicksort, the modular base will consist exactly of partitions for which the elements consist of the lists which share the same pivot element.

2. Background

2.1. Compositionality of timing measures

Let $A; B$ represent the sequential execution of algorithm A followed by algorithm B, where A operates on the input multiset $\mathcal{I}$ and produces the output multiset $A(\mathcal{I})$.

The worst-case time satisfies the following compositionality inequality:

$$(*)\ T^W_{A;B}(\mathcal{I}) \leq T^W_A(\mathcal{I}) + T^W_B(A(\mathcal{I}))$$

The average-case time satisfies the following compositionality equality [18, 22].

$$(**)\ \overline{T}_{A;B}(\mathcal{I}) = \overline{T}_A(\mathcal{I}) + \overline{T}_B(A(\mathcal{I}))$$

For the average-case time measure, we focus on finite input sets. This corresponds to conventions of traditional average-case analysis. Indeed, for the discrete case, inputs are identified up to order isomorphism yielding a finite amount of "states" used during the analysis. For the case of lists of size n, the analysis is reduced to considering the $n!$ cases of input lists as opposed to the potentially infinite collection of inputs. The argument, as usual, relies on the assumption that the algorithm under consideration runs in the same time on lists that satisfy the same relative order between elements. This will be the case for the algorithms we consider.

We recall the proof from [18] (where the multiset $\mathcal{I}$ is finite):

$$\begin{aligned}
\overline{T}_{A;B}(\mathcal{I}) &= \frac{\sum_{I\in\mathcal{I}} T_{A;B}(I)}{|\mathcal{I}|} \\
&= \frac{\sum_{I\in\mathcal{I}} T_A(I) + \sum_{J\in\mathcal{O}_A(\mathcal{I})} T_B(J)}{|\mathcal{I}|} \\
&= \overline{T}_A(\mathcal{I}) + \overline{T}_B(\mathcal{O}_A(\mathcal{I})),
\end{aligned}$$

where the last equality follows from the fact that $|\mathcal{I}| = |\mathcal{O}_A(\mathcal{I})|$ (where the collections involved are multisets).

Even though the average-case time exhibits a nice compositionality property (an equality as opposed to a bound for the worst-case time), determining the multiset $\mathcal{O}_A(\mathcal{I})$ is a non-trivial problem. The lack of an efficient method to track data has plagued static average-case analysis approaches. This is illustrated by open problems in the average-case analysis of algorithms, such as the exact average-case analysis of traditional Heapsort commented on by Knuth in [9]. Related issues also arise for Knotts paradox for binary search trees [9]. The root of the problem lies in the fact that these algorithms are not randomness preserving (as pointed out in [9] for the selection phase of

Heapsort. This has made the exact average-case analysis of all Heasport variants impossible [6]. A resolution to the problem was obtained in [18], where a randomness preserving version of Heapsort, Percolating Heapsort, has been obtained for which the exact average-case analysis has been determined. For a history of these problems we refer the reader to [18]. For randomness preserving algorithms however, it is possible to represent the multiset $\mathcal{O}_A(\mathcal{I})$ as a random bag. For such randomness preserving algorithms, the following compositionality theorem holds.

Theorem 2. *(Compositionality Theorem, [18, 22]) Consider random bag preserving programs/operations P and Q, where we execute P on a random bag R, producing random bag R'.*

- *The average-case time of the sequential execution of P followed by Q is:*
$$\overline{T}_{P;Q}(R) = \overline{T}_P(R) + \overline{T}_Q(R').$$
- *Consider random bag $R = \{(R_1, K_1), \ldots, (R_n, K_n)\}$, then:*
$$\overline{T}_P(R) = \sum_{i=1}^{n} Prob_i \times \overline{T}_P(R_i)$$
where
$$Prob_i = Prob[F \in R_i] = \frac{K_i|R_i|}{\sum_{i=1}^{n} K_i|R_i|} = \frac{K_i|R_i|}{|R|}$$
where F is any labeling belonging to the random structure R_i.
- *For the particular case where $R = \{(R_1, K_1)\}$, the previous equality reduces to:*
$$\overline{T}_P(R) = \overline{T}_P(R_1).$$

The compositionality theorem will be used to derive the modular smoothed complexity of Quicksort.

The compositionality theorem has been fruitfully applied to design new static average-case timing tools. We give a brief overview of the approach. For the purposes of the present paper however, the reader interested in the modular approach to smoothed complexity can omit reading the next part and skip to Section 2.2.

The compositionality problem for average-case analysis has been addressed via the $\mathcal{MOQA}$[10] language. No prior knowledge of the language is needed for this paper. For an introduction to the $\mathcal{MOQA}$ language and the (semi-)automated derivation of the average-time, we refer the reader to [18].

The fundamental concepts underlying the approach are the notion of random bags and their preservation. These concepts intuitively capture the data distribution and its preservation. The $\mathcal{MOQA}$ language essentially consists of a suite of data-structuring operations together with conditionals, for-loops and recursion. As such $\mathcal{MOQA}$ can be incorporated in any traditional programming language, importing its benefits in a familiar context.

In a nutshell, $\mathcal{MOQA}$ enables the finitary representation and tracking of the distribution of data states throughout computations, supporting compositional reasoning [18, 22]. This approach has been developed for discrete algorithms, in particular the class of random bag preserving comparison-based algorithms. The class encompasses many sorting and search algorithms [18] and hence forms a useful testing ground to explore a modular approach to smoothed complexity in the discrete case.

The time analysis for these algorithms has been given in [18].

The tracking of the data states is achieved through a finitary representation of the distribution via a random bag[11] and through a careful design of the basic operations all of which are random bag preserving. The static average-case time is derived by relying on the Compositionality Theorem.

An extension of the $\mathcal{MOQA}$ language has been shown to be Turing complete in [28].

$\mathcal{MOQA}$ offers a guaranteed average-case timing compositionality. It enables the prediction of the average number of basic steps performed in a computation.

$\mathcal{MOQA}$ gave rise to new algorithms, such as the sorting algorithm Percolating Heapsort, resolving the open problem on the exact analysis of Heapsort variants [18, 21] and the algorithm Treapsort, for which a smoothed analysis has been carried out in [27]. Applicatons of $\mathcal{MOQA}$ to reversible computing, data structures and parallel computation are the topic of [3, 24, 23]. Similar to the usefulness of a (partial) compositionality principle in a WCET context, the availability of a compositionality principle for average-case time has

[10] MOdular Quantitative Analysis

[11] As observed we use the term bag and multiset interchangeably.

paved the way for static average-case timing tools. Two timing tools have been explored to date: Distri-Track [18, 7] and a domain specific language interpretation for $\mathcal{MOQA}$ relying on abstract evaluation to derive the timing information. [2].

Here, we focus on exploring modular smoothed analysis, applied to the well-known Quicksort algorithm.

2.2. Smoothed complexity and the partial permutation model

Smoothed analysis considers inputs that are subject to some random permutation. The *smoothed measure* of an algorithm acting on a given input is the average running time of the algorithm over the perturbations of that instance, while the *smoothed complexity* of the algorithm is the worst smoothed measure of the algorithm on any input instance. The degree of perturbation is measured by a parameter σ. As σ becomes very small, the perturbations on the input become insignificant, and the smoothed complexity tends towards the worst-case running time. As σ becomes large, the perturbations become more significant than the original instance and the smoothed complexity tends towards the average-case running time.

In general, the smoothed complexity is a function of σ which interpolates between the worst case and average case running times. The dependance on σ gives a sense of how improbable an occurrence of the worst case input actually is. Formally, we have the following definition for smoothed complexity:

Definition 1. [16] Given a problem P with input domain $\mathcal{D} = \mathcal{U}_n\mathcal{D}_n$ where $\mathcal{D}_n$ represents all instances whose input size is n. Let $\mathcal{R} = \bigcup_{n,\sigma} \mathcal{R}_{n,\sigma}$ be a family of perturbations where $\mathcal{R}_{n,\sigma}$ defines for each $x \in \mathcal{D}_n$ a perturbation distribution of x with magnitude σ. Let A be an algorithm for solving P. Let $T_A(x)$ be the complexity for solving an instance $x \in \mathcal{D}_n$.[12]
The smoothed complexity: $T_A^S(n)$ of the algorithm A is defined by:

$$T_A^S(n, \sigma) = max_{x \in \mathcal{D}_n} \big(\mathbb{E}_{y \leftarrow \mathcal{R}_{n,\sigma}(x)}[T_A(y)]\big),$$

where $y \leftarrow \mathcal{R}_{n,\sigma}(x)$ means y is chosen according to distribution $\mathcal{R}_{n,\sigma}(x)$. The smoothed complexity is the worst of smoothed measures of A on inputs of

[12] In our context: The algorithms will be comparison based and $T_A(x)$ will be the running time of A input x, measured in the number of comparisons A will carry out when computing the output on input x.

size n of the expected value (average time) of algorithm A on the family of perturbations of x, namely the set $\mathcal{R}_{n,\sigma}(x)$ = smoothed complexity measure.

The method of partial permutations to study the smoothed complexity of discrete data was first proposed in [4], who defined it as follows:

Definition 2. [4] *Partial Permutations:* This model applies to problems defined on sequences. It is parameterized by a real parameter σ with $0 \leq \sigma \leq 1$ and is defined as follows. Given a sequence $s_1, s_2, \ldots s_n$ each element is selected (independently) with probability σ. Let k be the number of selected elements (on average $k = \sigma n$). Choose one of the $m!$ permutations of m elements (uniformly at random) and let it act on the selected elements.

Example 1. [4] For $\sigma = 1/2$ and $n = 7$, one might select $m = 3$ elements(namely s_2, s_3 and s_7) out of an input sequence $(s_1, \overline{s_2}, s_3, \overline{s_4}, s_5, s_7\overline{s_7})$. Applying the permutation (312) to the selected elements yields

$$(s_1, \overline{s_7}, s_3, \overline{s_4}, s_5, s_6\overline{s_4}).$$

As stated in the introduction, the natural model of permutation on a sequence for recursive algorithms applies partial permutations at each call of the recursive algorithm. The input partitions and their perturbations are tracked throughout the computation and the newly produced outputs will be perturbed yet again, and passed on to the next basic operation involved in the algorithm. For this reason, we modify the partial permutation model defined in [4] and define a new model. We refer to our model as the recursive partial permutation model, and define it as follows:

Definition 3. *Recursive Partial Permutations:* This model applies partial permutations at each successive call of the recursive algorithm.

In $\mathcal{MOQA}$ we typically assume that all data (LPO = labelled partial order) has been created from the atomic random structures $\mathcal{A}_n (n \geq 1)$ These random structures can be represented (after identification up to label isomorphism) as the collection of permutations of the first n integers, denoted by $\sum_n$ [13]

[13]Note: $\mathcal{MOQA}$ programs can operate on arbitrary random structures or random bags, provided certain rules are respected. We consider in first instance $\mathcal{A}_n = \sum_n$ in particular since we analyze quicksort whose inputs stem from $\sum_n$.

For the case of $\sum_n$ we can carry out the following simplification. Definition 2 can be simplified to a random selection of k elements among an input permutation of size n, where for sufficiently large n the number of selected elements k will be close to $n\sigma$, i.e. $\frac{n}{k} \approx \sigma$.

In essence, this simplification amounts to focusing on the expected outcome of selecting the elements with probability σ, which in case of an outcome of k elements is $\frac{k}{n}$, where the expected outcome is a selection of k elements (and other outcomes become negligible in chance.) The formalization can be based on a similar argument as is presented in [13] relying on Chernoff bounds. The motivation for relying in our arguments on this simplification is that considering σ to be of the form $\frac{k}{n}$ allows for the expression of the smoothed complexity via a recurrence equation in terms of n and k.

Taking account of the above, from here on we focus, for inputs of size n from $\sum_n$, on probabilities $\sigma = \frac{k}{n}(k \geq 0, k \leq n)$ and on partial permutations (perturbations of magnitude σ). Our definition of partial permutations now becomes:

Definition 4. A σ-partial permutation of s is a random sequence $s' = (s'_1, s'_2, \ldots s'_n)$ obtained from $s = (s_1, s_2, \ldots s_n)$ in two steps.

1. k elements of s are selected at random, where $k \geq 0, k \leq n$.
2. Choose one of the $k!$ permutations of these elements (uniformly at random)and rearrange then in that order, leaving the positions of all the other elements fixed.

We now adapt our notation in definition 1 according to our new definition above:

For $\sigma = \frac{k}{n}$, $\mathcal{R}_{n,\sigma}$ can be denoted as $\mathcal{R}_{k,n}$, the collection of partial permutations of size n that permute k out of n elements and leave the others fixed. If $s \in \sum_n$ and $t \in \mathcal{R}_{k,n}(\mathcal{R}_{k,n}$ will also be denoted as $\sum_{k,n})$ then $t \circ s$ denotes the effect of carrying out the partial permutations t on the permutation s.

The average time $\overline{T}$ of an algorithm A on an input collection $\mathcal{I} \subset \mathcal{D}_n$ (in our case $\mathcal{D}_n = \sum_n$) is

$$\overline{T}_A(\mathcal{I}) = \frac{\sum_{i \in \mathcal{I}} T_A(i)}{|\mathcal{I}|}.$$

$$\overline{T}_A(n) = \overline{T}_A(\mathcal{D}_n) = \frac{\sum_{s \in \sum_n} T_A(s)}{n!} \tag{3}$$

Table 1: Partial permutations for $n = 3$ on the set of permutations of $\{1, 2, 3\}$.

$\sum_3$	$\sum_{1,3}$	$\sum_{2,3}$	$\sum_{3,3}$
123	123, 123, 123	123, 123, 123	132, 231, 312
		132, 213, 321	123, 213, 321
132	132, 132, 132	132, 132, 132	132, 231, 312
		123, 312, 231	123, 213, 321
213	213, 213, 213	213, 213, 213	132, 231, 312
		231, 123, 312	123, 213, 321
231	231, 231, 231	231, 231, 231	132, 231, 312
		213, 321, 132	123, 213, 321
312	312, 312, 312	312, 312, 312	132, 231, 312
		321, 132, 213	123, 213, 321
321	321, 321, 321	321, 321, 321	132, 231, 312
		312, 231, 123	123, 213, 321

The definition of the smoothed complexity now simplifies:

$$T_A^S(n,k) = max_{s \in \sum_n}(\overline{T}_A(Pert_{k,n}(\mathcal{A}))) \tag{4}$$

where $Pert_{k,n}(\mathcal{A}) = \{t \circ s | s \in \mathcal{A}, t \in \sum_{k,n}\}$ is a multiset.

Lemma 3. $Pert_{k,n}(\sum_n) = \left\{ \left(\sum_n, \binom{n}{k} k!n! \right) \right\}$

Proof. The proof is left as an exercise. □

Let us now look at the two extreme cases of perturbations, that is, when $k = 1$ and $k = n$.

Example 2. For $n = 3$, we consider partial permutations ("1-selections") on $\sum_3 = \{(123), (132), (213), (231), (321)\}$. For each input there are three possible selections of 1 elements. The only permutation on 1 element is the identity. Hence, $Pert_{1,3}(213) = \{(213), (213), (213)\}$

$$\overline{T}_A(Pert_{1,3}(213)) = \frac{\sum_{i=1}^{3} T_A(213)}{3} = T_A(213)$$

so in general $T_A^S(n, 1) = max_{s \in \sum_n} T_A(s) = T_A^W(n)$, T_A^W being the worst case time for the algorithm A.

Example 3. For each $s \in \sum_n Pert_{n,n}(s) = \{t \circ s | t \in \mathcal{R}_{n,n} = \sum_n\} = \sum_n$ so

$$T_A^S(n,n) = max_{s \in \sum_n}(\overline{T}_A(\sum\nolimits_n)) = \overline{T}_A(n)$$

$\overline{T}_A$ being the average case time for the algorithm A.

3. Sample-Fair Smoothed Complexity

To set the stage for our main argument, we discuss the notion of sample-fairness. Here we consider the following generalization of the smoothed complexity, which includes all possible smoothed measures. Formally, we take the maximum over the set of partitions of $\sum_n$, $\mathcal{P} = Partitions(\sum_n)$, where $P \in \mathcal{P}$ is denoted by $P = \{P_1, \ldots P_m\}$ and $i \leq m \leq n!$. We define the smoothed complexity of a partition $\mathcal{P}, P = \{P_1, \ldots, P_m\}$ of $\sum_n$ as $T_A^S(P,k) = max_{i \in \{1,\ldots,m\}} \overline{T}_A(Pert_{k,n}(P_i))$.

We define the sample-fair smoothed measure as follows:

$$T_A^{SFS}(n,k) = max_{P \in \mathcal{P}, P = \{P_1,\ldots,P_m\}}[T_A^S(P,k)] \tag{5}$$

Smoothed complexity is the restriction of the fair smoothed complexity to a single partition P of $\sum_n$ consisting of all singletons, $P = \{\{s_1\}, \ldots, \{s_{n!}\}\}$, which we refer to as the *singleton base* in the following. Hence $T_A^{SFS}(n,k) \geq T_A^S(n,k)$.

Even though the fair smoothed complexity seems to take into account more sample spaces and hence may yield a different result, we will show that this is not the case.

Theorem 4. $T_A^{SFS}(n,k) = T_A^S(n,k)$.

Proof. As observed above, $T_A^{SFS}(n,k) \geq T_A^S(n,k)$, hence it suffices to show that $T_A^{SFS}(n,k) \leq T_A^S(n,k)$, or:

$$\forall P \in \mathcal{P} = Partitions(\sum\nolimits_n).\, T_A^S(P,k) \leq T_A^S(n,k),$$

where $P = \{P_1, \ldots, P_m\}$.

Hence we will show: $\forall i \in \{1, \ldots, m\}.\, \overline{T}_A(Pert_{k,n}P_i) \leq T_A^S(n,k)$.

Let $P_i = \{s_{j_1}, \ldots, s_{j_{c(i)}}\}$, where $c(i) = |P_i|$, then

$$\overline{T}_A(Pert_{k,n}P_i) = \overline{T}_A(Pert_{k,n}(s_{j_1}) \uplus \ldots \uplus Pert_{k,n}(s_{j_{c(i)}}))),$$

where $\uplus$ is the multiset union.

For $l \in \{1, ..., c(i)\}$, let $Pert_{k,n}(s_{j_l}) = \{s^l_1, \ldots, s^l_{f(l)}\}$.

Then:

$$\begin{aligned}
\overline{T}_A(Pert_{k,n}P_i) &= \overline{T}_A(Pert_{k,n}(s_{j_1}) \uplus \ldots \uplus Pert_{k,n}(s_{j_{c(i)}})) \\
&= \frac{\sum_{v=1}^{f(1)} T(s^1_v) + \ldots + \sum_{v=1}^{f(c(i))} T(s^{c(i)}_v)}{\sum_{l=1}^{c(i)} f(l)} \\
&= \frac{f(1)\frac{\sum_{v=1}^{f(1)} T(s^1_v)}{f(1)} + \ldots + f(c(i))\frac{\sum_{v=1}^{f(c(i))} T(s^{c(i)}_v)}{f(c(i))}}{\sum_{l=1}^{c(i)} f(l)}
\end{aligned}$$

Let $max(\overline{T}(Pert_{k,n}(s_{j_1})), \ldots \overline{T}(Pert_{k,n}(s_{j_{c(i)}}))) = \overline{T}(Pert_{k,n}(s_{j_t}))$ for some $t \in \{1, \ldots, c(i)\}$, then:

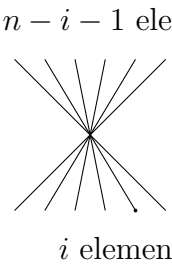
$$\begin{aligned}
\overline{T}_A(Pert_{k,n}P_i) &\leq \frac{f(1)\overline{T}(Pert_{k,n}(s_{j_t})) + \ldots + f(c(i))\overline{T}(Pert_{k,n}(s_{j_t}))}{\sum_{l=1}^{c(i)} f(l)} \\
&= \overline{T}(Pert_{k,n}(s_{j_t})).
\end{aligned}$$

QED $\square$

The singleton base has been shown to suffice for a fair representation of all smoothed measures, however for the purpose of this paper, Theorem 4 allows to us to consider an alternative partition as our base, without any abuse of terminology. As motivated in the introduction, the modular smoothed complexity has been formulated to reflect "modular-fairness", where perturbations are systematically applied to all inputs of each basic operation of an algorithm, rather than to the algorithm's original inputs only. In the following we will illustrate how the choice of a different base can be used to define the notion of a modular smoothed complexity $T^{MS}_A(n, k)$. Intuitively, the modular smoothed complexity is defined similarly to the traditional smoothed complexity, for the case of recursive algorithms, with the distinction that the definition uses an alternative partition, referred to as the *modular base*. We will formalize the approach below.

4. Modular smoothed complexity of Quicksort

First, we recall some basic facts about the $\mathcal{MOQA}$ split operation, the central operation in Quicksort. For more details, we refer the interested reader to [22]. When split is executed on the labelings of the discrete order of size n, it produces a random bag for which each partial order is order isomorphic to one of the orders $P[i,j]$ (where $n = i+j+1, i, j : 0, \ldots n-1$) of the shape



We assume the pivot to be the first list element

$$Split : \mathcal{R}(\Delta_n) \longrightarrow \{R(P[n-1,0], K_0), \ldots R(P[0,n-1], K_n)\}$$

where $K_i = \binom{n-1}{i}, i = 0, \ldots n-1$

Note that the Random bag $\{(R_0, K_0), \ldots (R_n, K_n)\}$ (where $R_k = R(P[k, n-1])$) is a uniform random bag where

$$K_0.|R_0| = K_1.|R_1| = \ldots K_n.|R_n|.$$

See [18] for further details.

Next we analyze the effect of carrying out the split operation on partial permutations of inputs. Firstly some observations. In [4] it is shown that partial permutations on inputs do not have the typical property that recursive calls of quicksort(ie Split) result in subproblems that are again random permutations (See "pitfalls" on page 4 in [4]). For this reason we will focus on identifying collections of inputs on which partial permutations are guaranteed to lead to random subproblems on which split can be called. These random subproblems will be random bags. This is a generalization of the definition of smoothed complexity, where we will consider a partition of the inputs $\sum_n$ say $P_1, \ldots P_l$, $(P_1 \cup \cdots \cup P_l) = \sum_n$ and $\forall i, j, i \neq j \implies P_i \cap P_j = \phi$.

We will generalize the smoothed complexity as follows:

$$T_A^S(n,k,P) = max_{i \in \{1,\ldots,l\}}(\overline{T}_A(Pert_{k,n}(P_i))) \tag{6}$$

We obtain the standard smoothed complexity by picking the partition $P_i = \{s_i\}, i \in \{1, \ldots n!\}$ (partition of singleton sets of $\sum_n$)

Each random bag preserving $\mathcal{MOQA}$ operation ϕ can be viewed as a transformation (function) from a random structure to a random bag[14].

$$\phi : R \longrightarrow \{(R_1, K_1), (R_2, K_2), \ldots (R_n, K_n)\}$$

The definition of Random Bag preservation means that there exists a partition $\{P_1, \ldots P_n\}$ of R such that

$$\phi : P_i \longrightarrow \{(R_i, K_i)\}$$

Definition 5. The partition $\{P_i, \ldots P_k\}$ as defined above is called "the base" of the operation ϕ

Hence for $\mathcal{MOQA}$ operations with base $P = \{P_1 \ldots P_k\}$ we can define the smoothed complexity as in Eq. 6.

$$T_\phi^{MS}(n, k, P) = max_{i \in \{1, \ldots l\}} \overline{T}_A(Pert_{k,n}(P_i)) \tag{7}$$

where ϕ is a random bag preserving operation and $\{P_1, \ldots P_l\}$ is a base for the operation. We note that we will later adapt this for the purpose of our recursive calculation of Quicksort.

As split is the central operation in Quicksort we will now focus on computing $T_{split}^{MS}(n, k)$. Firstly we re state that the pivot of split is the first list element (first elements of input sequence $s \in \sum_n$) The base of split, as referred to in the last section is the partition $(\sum_n^i)_{i \in \{1, \ldots n\}}$ where

$$P_i = \sum\nolimits_n^i = \{s \in \sum\nolimits_n | s_1 = i\}$$

We will show that split is a random bag preserving operation on each perturbation of $\sum_n^i$(base element). As $\sum_n^i$ strictly speaking yields a set of perturbations that is not immediately representable as a random bag, we abuse our terminology. Instead we will relax the terminology as follows: An operation A is "random bag inducing" on a collection of inputs I when the output multiset yields a random bag R, $A : I \to R$. We now present the following theorem:

[14]The operation ϕ can be extended to random bags by having the operation act on each random structure in the random bag [18, 22].

Theorem 5. *Split is random bag inducing on each of its perturbations of base elements i.e on each multiset* $Pert_{k,n}(\sum_n^i)$ *where* $i \in \{1, \dots n\}$. *More precisely,*
Split: $Pert_{k,n}(\sum_n^i) \longrightarrow \{(R(P[n-1,0]), K_1^i), \dots (R(P[0,n-1]), K_n^i)\}$
where

$$K_j^i = \frac{(n-2)!(k-1)}{(n-k)!}\binom{n-1}{j-1}, i \neq j \tag{8}$$

$$K_i^i = \frac{(n-1)!(n-k+1)}{(n-k)!}\binom{n-1}{i-1}, i = j \tag{9}$$

Our counting argument can be found in Appendix A

We note that in the rest of this paper, we use an abbreviation recording only the multiplicities and shorthand R_k for $R(P[k-1, n-k])$. It is clear from above that split on $Pert_{k,n}(\sum_n^i)$ produces non-uniform random bags. Now that we have established the effect of split on our base elements, we now return to solving the following equation:

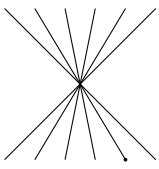
$$T_{QS}^{MS}(n,k) = max_{i \in \{1 \dots n\}}(\overline{T}_{QS}(Pert_{k,n}(\sum_n^i))) \tag{10}$$

We note that we will follow the approach as presented in [21]. Firstly, we recall some of the notation needed in this section.

QS(X) = Split(X); Quicksort(Y_1); Quicksort(Y_2)

X stores labelings from $R(\Delta_n)$ i.e elements from $\sum_n$. Y_1 stores labelings that correspond to the restriction of the labeling Split(X) to I^{Lower}, where I^{Lower} is the lower part of $R_j = R(P[j-1, n-j])$. Similarly Y_2 stores the restriction of Split(X) to I^{Upper}. Split(X) is a labeling on $P[j-1, n-j]$.

$n-j$ elements(I^{Upper})



$j-1$ elements(I^{Lower})

We have the following lemma from [21]

Lemma 6. *When X ranges over $\sum_n$, the multiset of restrictions of Split(X) to I^{Lower} is a Random Bag*

$$\{R(\Delta_{j-1}), (j-1)!\} = \left\{ \sum\nolimits_{j-1}, (j-1)! \right\}$$

by a labeling isomorphism. Similarly for I^{Upper}:

$$\{(R(\Delta_{n-j}), (j-1)!)\} = \left\{ \sum\nolimits_{n-j}, (j-1)! \right\}.$$

We note here that we will break with the convention of that in eq. 10 where the recurrence equation is not a true recurrence, in the sense that the right hand side of the equation takes the maximum over the bases of the average running times after perturbation of the inputs. This approach holds with the conventional approach to Quicksort. However, in $\mathcal{MOQA}$ compositionality is key. For this reason we apply the *r*ecursive partial permutation model, so as to truly merge modularity and smoothed complexity. That is, we apply perturbations, not only on the inputs but on each atomic substructure involved in a recursive call. Thus, we now have

$$T_{QS}^{MS}(n,k) = max_{i\in\{1,\dots n\}} T_{QS}^{MS}\big(Pert_{k,n}(\sum\nolimits_n^i)\big)$$

where

$$\begin{aligned} T_{QS}\big(Pert_{k,n}(\sum\nolimits_n^i)\big) &= \overline{T}_{Split}\big(Pert_{k,n}(\sum\nolimits_n^i)\big) \\ &+ T_{QS}(\{(R(\Delta_0), N_i), \dots (R(\Delta_{n-1}), N_n^i)\}) \\ &+ T_{QS}(\{(R(\Delta_n), N_i), \dots (R(\Delta_0), N_1^i)\}) \end{aligned}$$

where $N_j^i = K_j^i L_j$ and $L_j = (n-j)!$.
We recall from Theorem 25 in [21] if $R = \{(R_1, K_1), \dots (R_p, K_p)\}$ is a random bag, we have

$$\overline{T}_p(R) = \sum\nolimits_{i=1}^{p} prob_i \overline{T}_p(R_i) \tag{11}$$

where

$$Prob_i = Prob[F \in R_i] = \frac{K_i|R_i|}{|R|}$$

We also note that split takes $n-1$ comparisons on every permutations of size n, thus:

$$\overline{T}_{Split}\big(Pert_{k,n}(\sum\nolimits_{n}^{i})\big) = n-1 \tag{12}$$

We now have $T_{QS}(Pert_{k,n}(\sum_{n}^{i}))$ defined as

$$max_{i\in\{1,\ldots n\}}\Bigg[(n-1)+\sum\nolimits_{j=1}^{n}\beta_{n+1-j}^{n}T_{QS}(\sum\nolimits_{j-1})+\sum\nolimits_{j=1}^{n}\beta_{j}^{i}T_{QS}(\sum\nolimits_{j-1})\Bigg]$$

where

$$\beta_{j}^{i}=\frac{N_{j}^{i}|\sum_{j-1}|}{|Pert_{k,n}(\sum_{n}^{i})|}$$

is the probability that a labeling belongs to $R(\Delta_{j-1})$ in the random bag $\{(R(\Delta_0),N_1^i),\ldots(R(\Delta_0),N_n^i)\}$ Recall, $|Pert_{k,n}|=|P_i^k|$ so for $i\neq j$ we have

$$\beta_{j}^{i} \;=\; \frac{N_{j}^{i}|\sum_{j-1}|}{|P_{i}^{k}|}=\frac{(k-1)}{n(n-1)}$$

and $i=j$ we have

$$\beta_{i}^{i} \;=\; \frac{N_{i}^{i}|\sum_{i-1}|}{|P_{i}^{k}|}=\frac{n-k+1}{n}$$

To summarize:

$$\begin{aligned}T_{QS}^{MS}(n,k) \;=\; & max_{i\in\{1,\ldots n\}}\Bigg[(n-1)+\sum\nolimits_{j=1}^{n}\beta_{n+1-j}^{i}T_{QS}^{MS}\Bigg(Pert_{k,n}(\sum\nolimits_{j-1})\Bigg)+\\ & \sum\nolimits_{j=1}^{n}\beta_{j}^{i}T_{QS}^{MS}\Bigg(Pert_{k,n}(\sum\nolimits_{j-1})\Bigg)\Bigg]\end{aligned}$$

where

$$\beta_{i}^{i}=\frac{n-k+1}{n}$$

and

$$\beta_{j}^{i}=\frac{(k-1)}{n(n-1)}$$

Let $f(n,k) = T_{QS}^{MS}(n,k)$. We now have the following recurrence equation

$$f(n,k) = max_{i\in\{1,\ldots n\}}\left[(n-1)+\sum_{j=1}^{n}\beta_{n+1-j}^{i}f(j-1,k)+\sum_{j=1}^{n}\beta_{j}^{i}f(j-1,k)\right] \tag{13}$$

Lemma 7.

$$max_{i\in\{1,\ldots n\}}\left[\sum_{j=1}^{n}\beta_{j}^{i}f(j-1,k)\right] = \sum_{j=1}^{n}\beta_{j}^{n}f(j-1,k)$$

Proof. The proof is left as an exercise □

The modular smoothed complexity of Quicksort, $f(n,k)$, is defined as,

$$f(n,k) = (n-1) + \sum_{j=1}^{n}\beta_{n+1-j}^{n}f(j-1,k) + \sum_{j=1}^{n}\beta_{j}^{n}f(j-1,k), \tag{14}$$

where

$$\beta_{n}^{n} = \frac{n-k+1}{n},$$

and

$$\beta_{i}^{n}[i \neq n] = \frac{(k-1)}{n(n-1)}.$$

5. Results and conclusion

Here we have presented a closed form equation for measuring the smoothed complexity of a randomness preserving algorithm. We have chosen Quicksort to illustrate this. From this equation we can find more precise bounds for the timing of such randomness preserving algorithms. Figure 2 shows the modular smoothed complexity of Quicksort for permutations of length $1 \leq n \leq 50$ for increasing values of the k. Figure 3 shows the timing for $n = 50, 100, 150$ and 200. In [2], the results presented in this paper were used to semi-automate the smoothed analysis of Quicksort. These results have been integrated into the interpreter analyzer for the $\mathcal{MOQA}$ language. Further details of this can be found in [2]

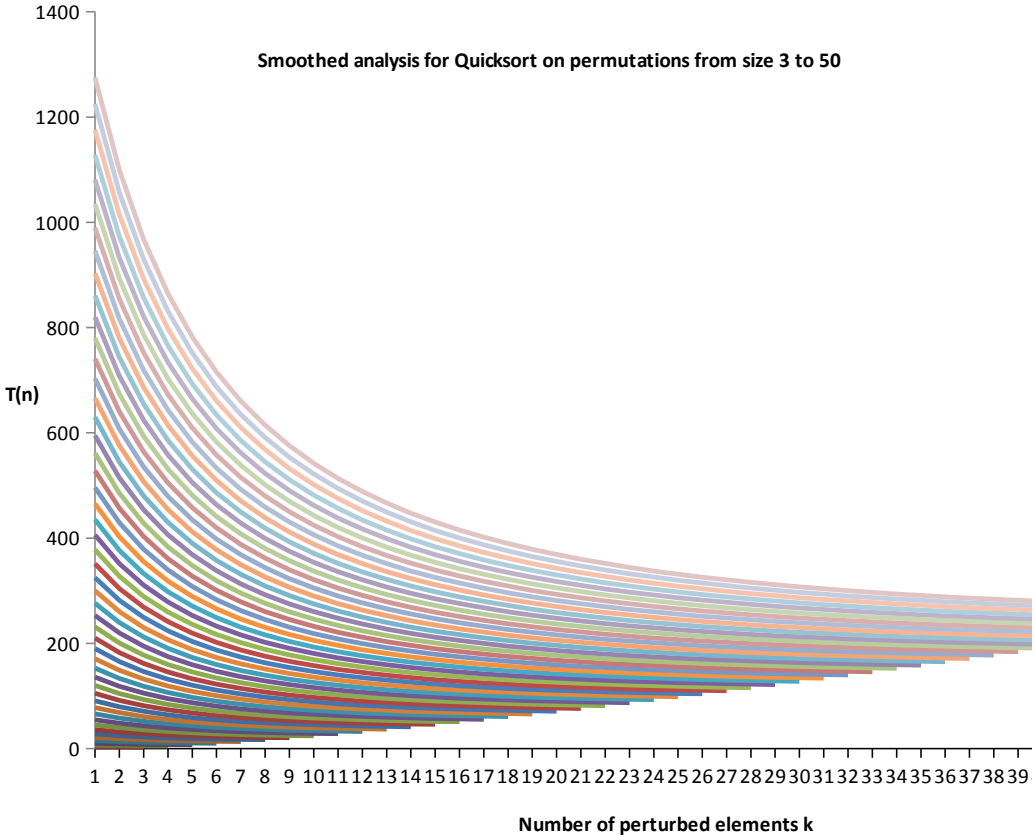


Figure 2: The modular smoothed complexity of Quicksort, for $1 \leq n \leq 50$, for increasing k

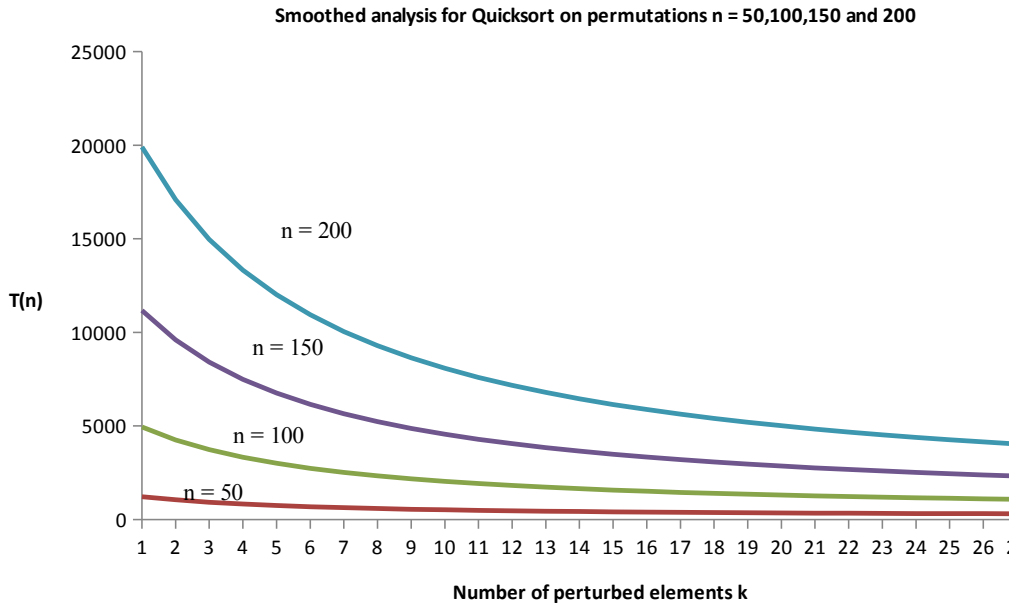


Figure 3: The modular smoothed complexity of Quicksort, for $n = 50, 100, 150$ and 200 for increasing k

6. Future work

As suggested in by Daniel Spielman in his commentary on [4], we will take a modular approach to median of three Quicksort, as an alternative to Quicksort. We compare and contrast these smoothed results with those presented above.

Appendix A.

Here we present counting arguments to prove eq. 8 and eq. 9 from Theorem 5. Firstly, we note that perturbations generated from a pivot will lead to more permutations with that pivot than any other pivot. This happens in all cases except where all elements are involved in the permutation(ie. $k = n$). All other pivots arise equally. For this reason, we proceed to count partial permutations of a permutation that keep the original pivot all other pivots separately.

Proof. Consider $\sum_n^i$ where each $s \in \sum_n^i$ is of the form $s = (i, s_2, \ldots s_n)$ so $\sum_n^i = \{i\} \sum_{n-1}$

Computation of K_j^i, where $j \neq i$:

Here we count the permutations with pivot j after permutation where pivot i is the pivot($i \neq j$) before perturbation. $Pert_{k,n}(\sum_n^j)$ can be counted as follows:

- Pivot i is always chosen as one of k perturbed elements(ie i has to move, for the pivot to become j).
- We count the permutations after perturbation that include the pivot i and j, where the choice is restricted for j, as this becomes the new pivot, resulting in $(k-1)!$ integers free, thus we count $\binom{n-2}{k-2}(k-1)!$.
- $(n-1)!$ permutations have the same pivot.

Following the above counting argument we now have

$$\begin{aligned} M_j &= \binom{n-2}{k-2}(k-1)!(n-1)! \\ &= \frac{(n-2)!(k-1)}{(n-k)!}(n-1)! \end{aligned}$$

A closer look at the previous argument shows that each sequence $s' \in \sum_n^j$ has the same chance to be formed. Hence, the multiset $Pert_{k,n}(\sum_n^i)$ contains $\frac{\sum_n^j M_j}{|\sum_n^j|}$. Split maps each $\sum_n^j$ so we obtain the bijection,
Split: $\sum_n^j \longrightarrow R(P[j-1, n-j])$, thus, for each $j \neq i$, the multiset $Pert_{k,n}(\sum_n^i)$ contains

$$\left\{ \left(R(P[j-1, n-j]), \frac{M_j}{(j-1)!(n-j)!} \right) \right\}$$

hence

$$K_j^i = \frac{M_j}{(j-1)!(n-j)!} = \frac{(n-2)!(k-1)}{(n-k)!}\binom{n-1}{j-1}.$$

Computation of K_i^i where $j = i$

For K_i^i we consider the random structure $P[i-1, n-i]$. Split will transform any element of $\sum_n^i$ to a labeling of $P[i-1, n-i]$ so we will determine how

many elements of the multiset $Pert_{k,n}(\sum_n^i)$ remain in $\sum_n^i$. If $s \in \sum_n^i$ then $s_1 = i$ so $s = \{i, s_2, \ldots s_n\}$.
The permutations after perturbation with pivot i can be counted as follows:

- Firstly, we count the permutations arising when the pivot is not chosen as one of the k integers to permute. These permutations are counted as $\left\{\binom{n}{k} - \binom{n-1}{k-1}\right\}k!$.
- When the pivot is chosen as one of the k integers for perturbation, we count the number of the permutations after perturbation that result in the pivot staying in place. This is counted by $\binom{n-1}{k-1}(k-1)!$.
- Both terms above are summed for the $(n-1)!$ permutations with the same pivot.

Following the above counting argument we now have

$$\begin{aligned} M_i &= \left(\left\{\binom{n}{k} - \binom{n-1}{k-1}\right\}k! + \binom{n-1}{k-1}(k-1)!\right)(n-1)! \\ &= \frac{(n-1)!(n-k+1)}{(n-k)!}(n-1)! \end{aligned}$$

Finally, each such sequence s gives rise to

$$\frac{(n-1)!(n-k+1)}{(n-k)!}(n-1)!,$$

so the multiset $Pert_{k,n}(\sum_n^i)$ contains

$$M_i = \frac{(n-1)!(n-k+1)}{(n-k)!}(n-1)!$$

elements of $\sum_n^i$.
We note that each of the above have the same probability of occurring, so once again

$$K_i^i = \frac{M_i}{(i-1)!(n-i)!} = \frac{(n-1)!(n-k+1)}{(n-k)!}\binom{n-1}{i-1}.$$

□

References

[1] A.Aho, J.Hopcroft and J. Ullman, Data Structures and Algorithms, Addison-Wesley Series in Computer Science and Information Processing, Addison-Wesley, 1987.

[2] A. Gao. Modular Average Case Analysis: Language Implementation and Extension, PhD thesis, University College Cork, 2013.

[3] A. Gao, K. Rea, and M. Schellekens, Static Average Case Analysis Fork-Join Framework Programs Based On $\mathcal{MOQA}$ Method, 6th International Symposium on Parallel Computing in Electrical Engineering, accepted for publication, Luton, UK, April 2011.

[4] C. Banderier, R. Beier, and K. Mehlhorn. Smoothed analysis of three combinatorial problems. In the 28th International Symposium on Mathematical Foundations of Computer Science, pages 198207, 2003.

[5] Hoare, C.A.R. Partition: Algorithm 63; Quicksort: Algorithm 64 and Find: ALgorithm 65 *Comm. ACM 4*, 7 (July 1961), 321–322.

[6] S. Edelkamp, Weak-Heapsort, ein schnelles sortierverfahren, Diplomarbeit Universität Dortmund, 1996.

[7] D. Hickey, Distritrack: Automated Average-Case Analysis, in the proceedings of the Fourth International Conference on the Quantatative Evaluation of Systems (QEST 2007), 17-19 September 2007, Edinburgh, Scotland, UK.

[8] D. Hickey, D. Early and M. Schellekens, A Tool for Average-Case and Worst-Case Execution Time Analysis, in proceedings of the Worst-Case Execution Time Workshop, satelite event of the Euromicro conference on Real-Time Systems, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (publisher), Germany, 2008.

[9] D. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, 1998.

[10] M. Li, P. Vitanyi, *An introduction to Kolmogorov Complexity and its Applications*, Texts and Monographs in Computer Science, Springer Verlag, 1993.

[11] R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[12] T. Maibaum, Mathematical Foundations of Software Engineering: a roadmap, Proceedings of the Conference on The Future of Software Engineering, ICSE00, 161 - 172, 2000.

[13] M. Mitzenmacher, E. Upfal, Probability and Computing: Randomized Algorithms and Probabilistic Analysis, Cambridge University Press, Cambridge, 2005.

[14] R. Sedgewick, Implementing quicksort programs, Comm. ACM **21** (10), 847–857, 1978.

[15] R. Schaffer and R. Sedgewick, The Analysis of Heapsort, Journal of Algorithms **15**(1), 76–100, 1993.

[16] D. Spielman, S. Teng, Smoothed Analysis: Why The Simplex Algorithm Usually Takes Polynomial Time, Journal of the ACM, Vol 51 (3), pp. 385 - 463, 2004.

[17] D. Spielman, S. Teng, Smoothed Analysis of Algorithms and Heuristics, Foundations of Computational Mathematics Santander 2005, London Mathematical Society Lecture Note Series, no. 331, Cambridge University Press, 274 - 342, 2006.

[18] M. P. Schellekens,"A Modular Calculus for the Average Cost of Data Structuring", Springer book, published in August, 2008.

[19] M. P. Schellekens, $\mathcal{MOQA}$ Unlocking the potential of compositional average-case analysis, Journal of Logic and Algebraic Programming, Volume 79, Issue 1, January 2010, Pages 61-83.

[20] D. Spielman, Commentary on Smoothed Analysis of Three Combinatorial Problems, published electronically at http://www.cs.yale.edu/homes/spielman/SmoothedAnalysis, 2003.

[21] M. Schellekens, G. Bollella and D. Hickey. $\mathcal{MOQA}$ a Linearly-Compositional Programming Language for (semi-) automated Average-Case analysis, IEEE Real-Time Systems Symposium - WIP Session, 2004.

[22] M. Schellekens, $\mathcal{MOQA}$ Unlocking the potential of compositional average-case analysis, Journal of Logic and Algebraic Programming, Volume 79, Issue 1, January 2010, Pages 61-83.

[23] Diarmuid Early, Ang Gao and Michel Schellekens. "Frugal encoding in reversible $\mathcal{MOQA}$ a case study for Quicksort". 4th Workshop on Reversible Computation, Copenhagen, Denmark, 2012.

[24] Ang Gao, Aoife Hennessy, Michel Schellekens: "$\mathcal{MOQA}$ Min-Max heapify: A Randomness Preserving Algorithm". 10th International Conference Of Numerical Analysis And Applied Mathematics, Kos, Greece, 2012.

[25] Kopetz, H.; Fohler, G.; Grnsteidl, G. et al.: RealTime Systems Development: The Programming Model of MARS, in Proceedings of the International Symposium on Autonomous Decentralized Sys- tems, pp. 190 199, Kawasaki, Japan, March. 1993.

[26] Erik Yu-Shing Hu, Guillem Bernat, Andy Wellings, A Static Timing Analysis Environment Using Java Architecture for Safety Critical Real-Time Systems, Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, p. 0077, 2002.

[27] D. Early, M. Schellekens, Running time of the Treapsort algorithm, Theoretical Computer Science, in press, accepted manuscript, available online from Springer Marh 25, 2013 at: http://www.sciencedirect.com/science/article/pii/S0304397513002132.

[28] D. Early, A Mathematical Analysis of the $\mathcal{MOQA}$ language, PhD thesis, University College Cork, 2010.

[29] D. Spielman, Smoothed Analysis Homepage, http://www.cs.yale.edu/homes/spielman/SmoothedAnalysis/.