Electronic Notes in Theoretical Computer Science 1 (1995)
URL: http://www.elsevier.nl/locate/entcs/volume1.html 22 pages

The Smyth Completion: A Common Foundation for Denotational Semantics and Complexity Analysis.

M. Schellekens [1]

Departement of Mathematics
Carnegie Mellon University
Pittsburgh, PA 15213
e-mail: Michel.Schellekens@cs.cmu.edu

Abstract

The Smyth completion ([15], [16], [18] and [19]) provides a topological foundation for Denotational Semantics. We show that this theory simultaneously provides a topological foundation for the complexity analysis of programs via the new theory of "complexity (distance) spaces". The complexity spaces are shown to be weightable ([13],[8],[10]) and thus belong to the class of S-completable quasi-uniform spaces ([19]). We show that the S-completable spaces possess a sequential Smyth completion. The applicability of the theory to "Divide & Conquer" algorithms is illustrated by a new proof (based on the Banach theorem) of the fact that mergesort has optimal asymptotic average running time.

1 History and Related Work

Smyth in [15] and [16] has provided a topological framework for Denotational Semantics based on the theory of quasi-uniform spaces (Nonsymmetric Topology [6], [8]). This work has been continued in [18] and [19] by Sünderhauf, in the context of the topological quasi-uniform spaces (a category extending the quasi-uniform spaces). The theory of the Smyth completion (or S-completion) has as a typical application to Denotational Semantics: the topological completion of a quasi-uniform space ("representing" a partial order) to a topological quasi-uniform space (which "represents" a cpo). The class of S-completable (topological) quasi-uniform spaces consists of the quasi-uniform spaces whose S-completion is again a quasi-uniform space. The S-completable quasi-uniform spaces have been introduced and characterized by Sünderhauf ([18] and [19]) and were shown to have an S-completion quasi-unimorphic

[1] I thank Stephen Brookes, Dana Scott and Rick Statman for helpful suggestions on the presentation of the paper.

(i.e. "isomorphic" in the context of quasi-uniform spaces) to the bicompletion ([19]). The weightable quasi-pseudo-metric spaces have been introduced by Matthews in the context of the semantic analysis of data flow networks ([13],[14]). Kunzi and Vajner have continued the study of the weightable spaces ([8],[9]) and these spaces have been shown to be S-completable by Kunzi ([8]).

We introduce a new quasi-pseudo-metric on function spaces suitable for the complexity analysis of programs: "the complexity distance". The complexity spaces (i.e. the spaces equiped with the complexity distance) are shown to be weightable and thus in particular to be S-completable.

We obtain a sequential version of the Smyth completion for S-completable quasi-uniform spaces and show that the Divide & Conquer algorithms induce contraction maps on the sequential Smyth completion of the complexity spaces. As an application an alternative proof is presented (based on the Banach theorem) of the fact that mergesort has optimal asymptotic average time.

2 Introductory Notions

We give the standard definitions. For an introduction to the theory of quasi-uniform spaces and Nonsymmetric Topology we refer the reader to [6] and [8].

We use $\circ$ to denote the composition of relations; if R is a relation, then R^{-1} is its inverse, and Δ stands for the diagonal relation, i.e. $\Delta = \{(x,x) | x \in X\}$. $\mathbb{N}$ denotes the set of the natural numbers and $\mathbb{R}$ denotes the set of the reals. $\mathbb{N}_0 = \mathbb{N} - \{0\}$. For any set X, $\mathcal{P}(X)$ denotes the powerset of X.

A *filter* $\mathcal{F}$ on a set X is a subset of $\mathcal{P}(X)$ such that

1) $(\forall F, G \in \mathcal{F})\ F \cap G \in \mathcal{F}$,
2) $(\forall G \subseteq X)[(\exists F \in \mathcal{F}\ F \subseteq G) \Rightarrow G \in \mathcal{F}]$,
3) $\emptyset \notin \mathcal{F}$.

A *quasi-uniform space* $(X, \mathcal{U})$ is a space such that

1) $\mathcal{U}$ is a filter on $X \times X$
2) $(\forall U \in \mathcal{U})(\exists V \in \mathcal{U})\ V \circ V \subseteq U$,
3) $(\forall U \in \mathcal{U})\ \Delta \subseteq U$.

$\mathcal{U}$ is a *quasi-uniformity* on X, and $U, V, W, \ldots$ denote elements of $\mathcal{U}$, called *entourages.*
The space $(X, \mathcal{U})$ is *uniform* when moreover:

4) $(\forall U \in \mathcal{U})\ U^{-1} \in \mathcal{U}$.

$\mathcal{U}$ is referred to as the *uniformity* on X.

A *base* $\mathcal{B}$ for a (quasi-)uniformity $\mathcal{U}$ is a subset of $\mathcal{U}$, such that

$$(\forall U \in \mathcal{U})(\exists B \in \mathcal{B})\ B \subseteq U.$$

A function $f : (X, \mathcal{U}) \to (Y, \mathcal{V})$ is *quasi-uniformly continuous* iff $(\forall V \in$

$\mathcal{V})(\exists U \in \mathcal{U})\ f^2(U) \subseteq V$, where $f^2(x,y) = (fx, fy)$. A *quasi-unimorphism* is a bijection f between quasi-uniform spaces such that f and f^{-1} are quasi-uniformly continuous. The topology $\mathcal{T}(\mathcal{U})$ associated with a uniformity $(X, \mathcal{U})$ is the topology generated by the family of neighbourhood filters $\{U[x] | U \in \mathcal{U}\}_{x \in X}$.

A topology is T_0 iff for all pairs of different points x, y either there exists a neighborhood V of x such that $y \notin V$ or there exists a neighborhood W of y such that $x \notin W$. A topology is T_1 iff for all pairs of different points x, y there exist neighborhoods V of x and W of y such that $y \notin V$ and $x \notin W$. The following properties hold (cf. [6]):

- Given a quasi-uniform space $\mathcal{U}$, then $\cap \mathcal{U}$ is always a preorder (i.e. reflexive and transitive). This preorder is called the *preorder associated to* U, and is denoted by $\leq_{\mathcal{U}}$.
- The topology $\mathcal{T}(\mathcal{U})$ is T_0 iff $\cap \mathcal{U}$ is a partial order and is T_1 iff $\cap \mathcal{U} = \Delta$.

For any entourage U, we define $U^* = U \cap U^{-1}$. Given a quasi-uniformity $\mathcal{U}$, $\mathcal{U}^*$ is defined to be the uniformity generated by the base $\mathcal{B} = \{U^* | U \in \mathcal{U}\}$, i.e.

$$\mathcal{U}^* = \{U \mid U \subseteq X \times X \text{ and } (\exists B \in \mathcal{B})\ B \subseteq U\}.$$

A $\mathcal{U}^*$-*Cauchy sequence* $(x_n)_n$ is a sequence such that

$$(\forall U^*)(\exists n_o)(\forall m, n \geq n_o)\ x_m U^* x_n.$$

A quasi-uniform space $(X, \mathcal{U})$ is *bicomplete* iff the uniform space $(X, \mathcal{U}^*)$ is complete. A *bicompletion* of a quasi-uniform space $(X, \mathcal{U})$ is a bicomplete quasi-uniform space $(Y, \mathcal{V})$ which has a $\mathcal{T}(\mathcal{U}^*)$-dense subspace quasi-unimorphic to $(X, \mathcal{U})$. T_0 quasi-uniform spaces have a unique (up to quasi-unimorphism) T_0 bicompletion ([6]), indicated by "the bicompletion".

A function $d : X \times X \to \mathbb{R}^+$ is a *quasi-pseudo-metric* iff

1) d satisfies the triangle inequality: $(\forall x, y, z)\ d(x,y) + d(y,z) \geq d(x,z)$, and
2) $(\forall x)\ d(x,x) = 0$.

d is a *quasi-metric* when d also satisfies

3) $d(x,y) = 0 \Rightarrow x = y$.

Given a quasi-pseudo-metric d, the *induced metric* d^* is defined to be $d^*(x,y) = max\{d(x,y), d(y,x)\}$. Note ([6]): a quasi-pseudo-metric d *generates* a quasi-uniform space $\mathcal{U}_d$ via the countable base $\mathcal{B}_d = (B_n)_n$, where $B_n = \{(x,y) | d(x,y) < 1/2^n\}$. So,

$$\mathcal{U}_d = \{U \mid U \subseteq X, (\exists B_n \in \mathcal{B}_d)\ B_n \subseteq U\}.$$

The *associated preorder* $\leq_d$ is defined by: $x \leq_d y$ iff $d(x,y) = 0$. This preorder coincides with the preorder associated to $\mathcal{U}_d$. Given a quasi-pseudo-metric space (X, d), a *d-contraction map* $f : X \to X$ is a map such that $(\exists c < 1)(\forall x, y \in X)\ d(fx, fy) \leq c d(x,y)$.

3 The Sequential Completion

For uniform spaces, the construction of the topological completion can be obtained via the standard minimal Cauchy filter completion ([5]). It is well known that in the case of a uniform space with a countable base the filter completion can be carried out by the usual Cauchy sequence completion (a property denoted in [5] by "adequacy of sequences"). This includes the particular case of the Cauchy sequence completion of a metric space. A completion via sequences will be referred to as a "sequential completion".

For Nonsymmetric Topology, for example for the theory of quasi-uniform spaces and in particular for the S-completion, the question of the existence of sequential completions is not settled. The S-completion of a quasi-uniform space as presented in the literature ([18]) is obtained via the (from a topological point of view) non-standard "round S-Cauchy" filters . In [19] Sünderhauf obtains a reduction of this filter S-completion to the "Cauchy net completion". However there is no guarantee that this completion further reduces to a sequential completion. In [15], a sequential (non-topological) completion of quasi-uniform spaces is given (of which we will use the definition of the base on the completion). However this sequential completion has not yet been generalized in the literature to the topological context of the S-completion. We obtain a partial answer to the problem by assuming the extra condition of "S-completability" ([18] and [19]) on the quasi-uniform space. (This extra assumption will be justified below, as the spaces we will consider will be S-completable.) Recall that a quasi-uniform space (as part of the topological quasi-uniform spaces) is S-completable iff its completion in the class of all topological quasi-uniform spaces is again a quasi-uniform space. Under the S-completability assumption the S-completion of a quasi-uniform space is quasi-unimorphic to its bicompletion [19]. This reduces the problem to finding a sequential version of the bicompletion of quasi-uniform spaces. To the author's knowledge no such version is available in the literature[2]. Theorem 3.1 sets up the sequential bicompletion of quasi-uniform spaces with countable base. In Corollary 3.2 we obtain the sequential bicompletion for the case of the quasi-pseudo-metric spaces. We define a quasi-pseudo-metric inducing the quasi-uniformity on the completion from a given quasi-pseudo-metric inducing the original quasi-uniformity. It is this last result which will be used to obtain the application discussed under section 7. The general result given under Theorem 3.1 combined with the fact that (as we will show) the space used in the application of section 7 is S-completable, implies that this application is based on a sequential S-completion, and thus on the topological foundation of Denotational Semantics.

[2] A sequential bicompletion for *quasi-pseudo-metric* spaces has been obtained by A. Di Concilio in [4], as pointed out recently to the author by H.P. Kunzi. This result corresponds to our Corollary 3.2. which follows from the more general result given under Theorem 3.1.

Theorem 3.1 *Every T_0 quasi-uniform space $(X, \mathcal{U})$ with countable base has a sequential bicompletion $(\overline{X}, \overline{\mathcal{U}})$. The base set $\overline{X}$ is defined by:*

$$\overline{X} = \{(x_n)_n | \, (x_n)_n \text{ is a } \mathcal{U}^*\text{-Cauchy sequence } \}/\approx,$$

where

$$(x_n)_n \approx (y_n)_n \quad \Longleftrightarrow \quad \forall B_k^* \; (\exists n_0)(\forall m, n \geq n_0) \; x_m B_k^* y_n.$$

The quasi-uniformity $\overline{\mathcal{U}}$ is defined to be generated by the base elements:

$$\{(\overline{x}, \overline{y}) \mid (\exists \text{ representatives } (x_n)_n, (y_n)_n \text{ of } \overline{x}, \overline{y}) \\ (\exists \, n_o)(\forall m, n \geq n_o) \; x_m B_k y_n\}.$$

(*The definition of the base corresponds to the one given in* [15].)

Proof (Sketch) The details are lengthy so we only give a sketch. We show that the obtained sequential completion is a T_0 bicompletion, so by the uniqueness of T_0 bicompletions (up to quasi-unimorphism) it is quasi-unimorphic to the filter bicompletion as given in [6]. □

Corollary 3.2 *When $\mathcal{U} = \mathcal{U}_d$, there is a quasi-pseudo-metric $\overline{d}$ and a metric $\overline{d^*}$, defined by:*

$$\overline{d}([(x_n)_n], [(y_n)_n]) = \lim_{n \to \infty} d(x_n, y_n) \quad \text{and}$$

$$\overline{d^*}([(x_n)_n], [(y_n)_n]) = \lim_{n \to \infty} d^*(x_n, y_n),$$

such that $\overline{\mathcal{U}} = \mathcal{U}_{\overline{d}}$, and such that $\overline{d}^* = \overline{d^*}$. In particular: $(\overline{\mathcal{U}}_d)^* = (\mathcal{U}_{\overline{d}})* = \mathcal{U}_{\overline{d}^*} = \mathcal{U}_{(\overline{d^*})}$. □

Two results for the sequential bicompletion are obtained next: the extension theorem (a straightforward adaption of the extension theorem in [6] for the filter bicompletion, to the sequential completion) and the Banach theorem (for the classical version of the Banach theorem we refer the reader to [5]).

Theorem 3.3 (Extension Theorem) *Suppose $(X, \mathcal{U})$ is a T_o quasi-uniform space with a countable base, and let $f\colon (X, \mathcal{U}) \to (X, \mathcal{U})$ be a quasi-uniformly continuous function. Then f extends uniquely to a quasi-uniformly continuous function $\overline{f} : (\overline{X}, \overline{\mathcal{U}}) \to (\overline{X}, \overline{\mathcal{U}})$, defined by $\overline{f}[(x_n)_n] = [(f(x_n))_n]$. $\overline{f}$ is unique in the sense that any quasi-uniformly continuous function $\overline{g} : (\overline{X}, \overline{\mathcal{U}}) \to (\overline{X}, \overline{\mathcal{U}})$ which agrees with $\overline{f}$ on the constant sequences must coincide with $\overline{f}$.* □

Theorem 3.4 (Banach Theorem) *Given a T_o space $(X, \mathcal{U}_d)$, and a d-contraction map f, the function $\overline{f}$ obtained via the extension theorem is a $\overline{d}$-contraction map, and has a unique fixed point $FIX(\overline{f})$, given by: $FIX(\overline{f}) = (\lim_{k \to \infty}(\overline{f}^k[(x_n)_n]))$, where $[(x_n)_n]$ is an arbitrary element of $\overline{X}$. In particular, for each constant sequence $[(x)_n]$: $FIX(\overline{f}) = [(\overline{f}^n x)_n]$.* □

4 The Complexity Distance

Given a partial recursive function f, and a programming language $\mathcal{L}$, let $[f]$ be the set of all programs of $\mathcal{L}$ computing a partial recursive function which approximates f (in the usual pointwise ordering on partial functions). We will use $P, Q, \ldots$ in what follows to denote programs.

Recall that a complexity measure ([3],[11]) is a binary partial function $C(k,n)$ on $\mathbb{N}^2$ satisfying the Blum axioms ([3]):

1) $C(k,n)$ is defined iff the program with coding k converges on input n.
2) the predicate $C(k,n) \leq y$ is recursive.

So $C(k,n)$ represents the complexity of a program P (with code k) on input n.

We only use this abstract setting to introduce the complexity distance in its full generality. We will not use the recursion theoretic machinery connected with this axiomatization, so the reader who is not familiar with abstract complexity measures, can keep one measure in mind (for example the running time) while reading the results.

The output value undefined is indicated by $\perp$, and has infinite complexity. Let C_P denote the complexity (function) of a program P, that is the function $\lambda n.C(k,n)$ (where k is the code of P). We assume C_P to be non-zero on all inputs.

The intuition behind the "complexity distance" (defined below) between programs P and Q is that $d(P,Q)$ measures *relative* progress made in lowering the complexity by replacing P by Q.

Definition 4.1 Given $\mathcal{P} \subseteq [f]$, we define

$$d\colon \mathcal{P}^2 \to \mathbb{R}^+ \text{ by } d(P_1,P_2) = \sum_{n\geq 0} d_{1,2}(n)\frac{1}{2^n},$$

where

$$d_{1,2}(n) = \begin{cases} 0 & \text{when } C_P\,(n) \leq C_{P_2}(n), \text{ and} \\ \frac{1}{C_{P_2}(n)} - \frac{1}{C_P\,(n)} & \text{otherwise.} \end{cases}$$

Note that the distance is normalized by the factor $\frac{1}{2^n}$ to guarantee convergence of the series.

Lemma 4.2 *The complexity distance is a quasi-pseudo-metric.* □

Note that d is not necessarily a quasi-metric, that is there may be programs P and Q such that $d(P,Q) = 0$ and $P \neq Q$. For example, consider a language which supports assignments and a complexity measure counting each assignment. Consider any program P in the language, and obtain the program Q from P by adding a dummy assignment to P (that is an assignment to a variable not occuring in P). We have $d(P,Q) = 0$ and $P \neq Q$.

A more precise motivation for the definition of the distance is given below. We follow the intuition given above, that is we aim at measuring relative progress in complexity. Assume P and Q are programs such that on a given input n, $C_P(n) > C_Q(n)$. We obtain a relative measure of progress by replacing the absolute difference $C_P(n) - C_Q(n)$ by $(C_P(n) - C_Q(n))/C_P(n)$. However, progress from $C_P(n) = \infty$ to $C_Q(n)$ (a finite value) with this distance would (by taking "the limit") yield constant value 1, independently of $C_Q(n)$. The limit we have in mind is $lim_{k\to\infty}(C_{P_k}(n) - C_Q(n))/C_{P_k}(n)$, where P_k is the program obtained from P by limiting P to run on each input for at most k steps.

To be consistent with the wish to measure relative progress, we should distinguish between improvements from infinity to different finite values. This can be obtained by replacing the above quotient by $(C_P(n) - C_Q(n))/(C_P(n) * C_Q(n))$, which amounts to $(1/C_Q(n)) - (1/C_P(n))$ when C_P and C_Q are finite. This last expression is sensitive to differences in finite values of C_Q, when C_P is infinite. The distance $d_{1,2}(n)$ from a defined value $P_1(n)$ to an undefined $P_2(n)$ (note the directedness) is defined to be 0, as this is consistent with the idea that there is no progress in an increase in complexity, in particular in an increase to infinite complexity.

It is clear that d suffers from an indifference against increases in complexity. This is unavoidable as the nonsymmetry of d avoids the problem which occurs for the induced metric d^*, namely the loss of information of which program choice (P or Q) is progress. The metric d^* does not give any information, in the sense that from the value $d^*(P,Q)$ it is impossible to determine which program would be more efficient. For instance assume that the program Q is more efficient on all inputs than the program P. In that case $d(P,Q)$ and $d^*(P,Q)$ have exactly the same value, but the last measure does not indicate which program is more efficient, while the first measure provides this information by the fact that $d(Q,P) = 0$. A second motivation for the use of a nonsymmetric distance is given below.

Lemma 4.3 *In general $([f], d)$ is not T_0, but can be made T_0 by the identification of programs with same outputs and complexity. The resulting quotient space $[f]'$ equipped with the induced distance d', defined by $d'([P],[Q]) = d(P,Q)$ is T_0. d' is well-defined and is a quasi-pseudo-metric. The space $([f]', d')$ is in general not T_1 (equivalently, d' is in general not a quasi-metric).*$\Box$

Convention: the equivalence class notation for elements of the resulting quotient $([f]', d')$ will be dropped in what follows, in particular elements of $[f]'$ will still be called "programs" belonging to $[f]'$. We will also use (with abuse of notation) $([f], d)$ for the quotient $([f]', d')$, that is we assume the spaces to be quotiented.

5 S-Completability

Given $\mathcal{P} \subseteq [f]$, define $\mathcal{C}_\mathcal{P}$ to be the set of complexity functions corresponding to the representatives of the elements of $\mathcal{P}$. As the space $([f], d)$ is obtained by taking the quotient which identifies programs which have same outputs and complexity, there is a bijection between $\mathcal{P}$ and $\mathcal{C}_\mathcal{P}$. So it is clear that the complexity distance can be directly defined on $\mathcal{C}_\mathcal{P}$, that is the spaces $([f], d)$ and $(\mathcal{C}_\mathcal{P}, d)$ trivially are quasi-unimorphic. In what follows we will not distinguish between the two approaches. In fact we will make the following generalization and work on the general function space $(0, \infty]^{\mathbb{N}}$, containing the set of all possible complexity functions C_P (recall our restriction: $C_P(n) \neq 0.$). Note that we do not necessarily have that functions of this space actually are complexity functions of a program.

The following results are stated for the function space approach, that is we work with a space (X,d), where $X \subseteq (0,\infty]^{\mathbb{N}}$ and where

$$d(f,g) = \sum_{n\geq 0}\{(\frac{1}{g(n)} - \frac{1}{f(n)})\frac{1}{2^n}|f(n) > g(n)\}.$$

Note that d is still a quasi-pseudo-metric, and that its associated order is the *pointwise* order on the function space $(0,\infty]^{\mathbb{N}}$. This pointwise order associated to the complexity distance is essential in comparing programs with respect to complexity, as will become clear in the complexity analysis of mergesort presented below.

We can now give a second motivation for the use of the nonsymmetric distance. As both the pointwise order and the induced metric d^* are definable from the nonsymmetric distance d, there is no need to introduce the metric and the order separately, that is the introduction of the nonsymmetric distance suffices.

Definition 5.1 A quasi-pseudo-metric space (X,d) is *weightable* iff there exists a function $w : X \to \mathbb{R}^+$ such that $\forall x,y \in X : d(x,y) + w(x) = d(y,x) + w(y)$. The function w is called a "weighting function" and $w(x)$ is called the "weight" of x, we say "d is weighted via w".

Weightable spaces were introduced by Matthews ([13]) in the context of the study of the semantics of data flow networks. Kunzi and Vajner have continued their study ([8],[9]). We recall the following result by Kunzi:

Proposition 5.2 (Kunzi) *The weightable quasi-pseudo-metric spaces are S-completable.* □

Notation: given $P,Q \in [f], then \sum_n^{\diamond}$ is the sum ranging over all n such that $C_P(n) \diamond C_q(n)$, where $\diamond$ is one of the following orders: $<, >, \leq, \geq$.

Proposition 5.3 *The complexity distance d on $X \subseteq (0,\infty]^{\mathbb{N}}$ is weighted via the weighting function w defined by:*

$$(\forall f \in X)\ w(f) = \sum_n \frac{1}{f(n)}\frac{1}{2^n}.$$

Proof. $\forall P,Q \in [f]$:

$$\begin{aligned}
d(P,Q) + w(P) &= \sum_n^{>}(\frac{1}{C_Q(n)} - \frac{1}{C_P(n)})\frac{1}{2^n} + \sum_n \frac{1}{C_P(n)}\frac{1}{2^n}\\
&= \sum_n^{>}\frac{1}{C_Q(n)}\frac{1}{2^n} + \sum_n^{\leq}\frac{1}{C_P(n)}\frac{1}{2^n}\\
&= w(Q) - \sum_n^{\leq}\frac{1}{C_Q(n)}\frac{1}{2^n} + \sum_n^{\leq}\frac{1}{C_P(n)}\frac{1}{2^n}\\
&= w(Q) + \sum_n^{\leq}(\frac{1}{C_P(n)} - \frac{1}{C_Q(n)})\frac{1}{2^n}\\
&= w(Q) + \sum_n^{<}(\frac{1}{C_P(n)} - \frac{1}{C_Q(n)})\frac{1}{2^n}\\
&= w(Q) + d(Q,P)
\end{aligned}$$

□

The following S-completability result is an immediate corollary of proposition 5.2 and justifies the existence of the sequential S-completion $(\overline{X}, \overline{d})$ of the complexity spaces (X, d).

Corollary 5.4 *For any $X \subseteq (0, \infty]^{\mathbb{N}}$, the space (X,d) is S-completable.* $\Box$

6 Divide & Conquer Algorithms

"Divide & Conquer" algorithms solve a problem by recursively splitting it into subproblems each of which is solved separately by the same algorithm, after which the results are combined into a solution for the original problem. The Divide & Conquer strategy is an important widely applicable technique for designing efficient algorithms ([1]). The complexity C of a Divide & Conquer algorithm typically is the solution to a recurrence equation of the form $C(1) = c$ and $(\forall n > 1)\, C(n) = aC(\frac{n}{b}) + h(n)$, where $a > 1$ represents the number of subproblems a problem is divided into, e represents the size of each subproblem and $h(n)$ represents the time it takes to combine the subproblems of a problem of size n into the solution. Divide & Conquer algorithms usually are assumed to be total, which is the assumption we make from here on. In particular we work on the space $(0, \infty)^{\mathbb{N}_0}$ rather than on $(0, \infty]^{\mathbb{N}_0}$. Note that we exclude 0 as an argument, since the recurrence equation has a base case determined by 1 rather than by 0 (this will be convenient in the presentation of the application of section 7, but is not an essential requirement).

Comment: Since we assume that the complexity on any input is never zero, we should require that $c \neq 0$ in order to guarantee that $(\forall n \geq 1)\, C(n) \neq 0$. Instead of requiring this condition (which will actually be violated in the application discussed under section 7) we impose the following natural condition: for each Divide & Conquer algorithm S we require that its complexity function satisfies $C_S(1) = c$. That is, we assume that each algorithm is such that it has the same complexity on the base case. Whatever recursive function the program actually computes, we can assume that the program has a built in test for the base case and behaves the same on this input. We are only interested in differences in complexity caused by the *recursion.* (Because of our assumption on the shape of the recurrence relation, we only consider recursion with one base case.)

The assumption on the base case implies that the value $c = 0$ does not cause any problems. Indeed, by this assumption we have $d_{1,2}(1) = 0$ (Definition 4.1, case 1) and thus division by 0 does not occur. So we continue to work with $[0, \infty)^{\mathbb{N}_0}$. Let $[0, \infty)^{\mathbb{N}_0}_c = \{f \mid f \in [0, \infty)^{\mathbb{N}_0} \text{ and } f(1) = c\}$. We denote this set in what follows by $\mathcal{C}_c$. Define for $b \in \mathbb{N}$, $b > 1$,

$$\mathcal{C}_c|b = \{f' \mid f' \text{ is the restriction of } f \in \mathcal{C}_c \text{ to arguments } n = b^k, k \geq 0\}.$$

The following theorem establishes the fact that Divide & Conquer algorithms induce contraction maps on the complexity spaces. This opens up the way to applications of the Banach theorem. We given an example of such an application in Section 7 below.

Theorem 6.1 (Contraction Map Theorem) *Let Φ_ϵ be the functional in-*

duced on $\mathcal{C}_c|b$ by the recurrence equation $\mathcal{E}$ defined by $C(1) = c$ and $(\forall n > 1)$ $C(n) = aC(\frac{n}{b}) + h(n)$, that is:

$$\Phi_{\mathcal{E}}: \mathcal{C}_c|b \to \mathcal{C}_c|b, \quad \textit{where } \Phi_{\mathcal{E}} = \lambda f \lambda n.\ \textit{if } n = 1 \textit{ then } c \textit{ else } af(\frac{n}{b}) + h(n).$$

Then $\Phi_{\mathcal{E}}$ is a d-contraction map iff $a > 1$, in which case the contraction constant is $\frac{1}{a}$.

Proof.

$$d(\Phi_{\mathcal{E}}(f), \Phi_{\mathcal{E}}(g)) = \sum_{\{n=b^k|k\geq 0\}}^{>} \left(\frac{1}{\Phi_{\mathcal{E}}(g)(n)} - \frac{1}{\Phi_{\mathcal{E}}(f)(n)}\right) \frac{1}{2^n}.$$

Note that for $k = 0$, $\Phi_{\mathcal{E}}(f)(1) = \Phi_{\mathcal{E}}(g)(1) = c$, so:

$$\begin{aligned}
d(\Phi_{\mathcal{E}}(f), \Phi_{\mathcal{E}}(g)) &= \sum_{\{n=b^k|k\geq 1\}}^{>} \left(\frac{1}{ag(\frac{n}{b}) + h(n)} - \frac{1}{af(\frac{n}{b}) + h(n)}\right) \frac{1}{2^n} \\
&= \sum_{\{n=b^k|k\geq 1\}}^{>} \left(\frac{af(\frac{n}{b}) + h(n) - (dg(\frac{n}{b}) + h(n))}{(af(\frac{n}{b}) + h(n))(ag(\frac{n}{b}) + h(n))}\right) \frac{1}{2^n} \\
&\leq \sum_{\{n=b^k|k\geq 1\}}^{>} \left(\frac{a(f(\frac{n}{b}) - g(\frac{n}{b}))}{(af(\frac{n}{b}))(ag(\frac{n}{b}))}\right) \frac{1}{2^n} \\
&\leq \frac{1}{a} \sum_{\{n=b^k|k\geq 0\}}^{>} \left(\frac{f(n) - g(n))}{f(n)g(n)}\right) \frac{1}{2^n} \\
&\leq \frac{1}{a} d(f, g).
\end{aligned}$$

□

Note that a functional $\Phi_{\mathcal{E}}$ induced by a Divide & Conquer recurrence equation $\mathcal{E}$ is monotone on $\mathcal{C}_c|b$, that is $(\forall f, g \in \mathcal{C}_c|b)\, f \leq g \Rightarrow \Phi_{\mathcal{E}} f \leq \Phi_{\mathcal{E}} g$.

We conclude the section with an application of the Banach Theorem to a special kind of functionals: the "(complexity) improvers". The intuition is that an improver is a functional which corresponds to a syntactic transformation on programs and which satisfies the following property: the iterative applications of the transformation to a given program yield an improved program at each step of the iteration.

Definition 6.2 A functional Φ on $\mathcal{C}_c|b$ is an *improver* with respect to a function $f \in \mathcal{C}_c|b$ iff $(\forall n)\, \Phi^{n+1} f \leq \Phi^n f$.

Note that when Φ is monotone, to show that Φ is an improver with respect to a function f, it suffices to verify that $\Phi f \leq f$.

The following proposition plays a crucial role in obtaining the application of section 7.

Proposition 6.3 *A Divide & Conquer recurrence equation $\mathcal{E}$ has a unique solution. If f is the solution to $\mathcal{E}$, and $\Phi_{\mathcal{E}}$ is an improver with respect to some function g, then $f \leq g$.*

Proof. (Sketch) Let $\mathcal{E}$ be a Divide & Conquer recurrence equation.Note that $\mathcal{E}$ is always solvable (cf. [1]). If f is a solution of $\mathcal{E}$, then we have that $\Phi_{\mathcal{E}} f = f$

and thus $\overline{\Phi_{\mathcal{E}}}[(f)_n] = [(\Phi^n_{\mathcal{E}} f)_n] = [(f)_n]$, that is $[(f)_n]$ is a fixed point of $\overline{\Phi_{\mathcal{E}}}$. By Theorem 6.1, $\Phi_{\mathcal{E}}$ is a contraction map on $\mathcal{C}_c$,and by the Extension theorem $\Phi_{\mathcal{E}}$ extends uniquely to a contraction map $\overline{\Phi_{\mathcal{E}}}$ on $\overline{\mathcal{C}_c}$. So by the Banach theorem $\overline{\Phi_{\mathcal{E}}}$ has a unique fixed point and thus $\mathcal{E}$ has a unique solution.

By the Banach theorem we have: $FIX(\overline{\Phi_{\mathcal{E}}}) = [(\Phi^n_{\mathcal{E}} g)_n]$, and thus, again by uniqueness of fixed points, $FIX(\overline{\Phi_{\mathcal{E}}}) = [(\Phi^n_{\mathcal{E}} g)_n] = [(f)_n]$.

Since $\Phi_{\mathcal{E}}$ is an improver with respect to g, we have: $\forall n \geq 0.\, \Phi^n_{\mathcal{E}} g \leq_d g$. So in particular $\lim_n d(\Phi^n_{\mathcal{E}} g, g) = 0$, and thus $\overline{d}([(\Phi^n_{\mathcal{E}} g)n], [(g)_n]) = 0$. So we have that $\overline{d}([(f)_n], [(g)_n]) = 0$, or equivalently $f \leq_d g$. $\square$

7 An Application: Mergesort

We will demonstrate the applicability of the theory to the complexity analysis of sorting algorithms for the specific complexity measure of average running time. All sorting algorithms S are assumed to be *comparison based* ([1], [7]). Comparison based programs are programs such that (ultimately) all computation steps carried out by S on any input list have to be based on a comparison between list elements. For this class of algorithms a lower bound on the average time is known: $\overline{T}(n) \geq n log_2 n$ ([1], [7]). We will present a novel proof (based on the Banach theorem) of the well known result that the (comparison based) sorting program *mergesort* has optimal asymptotic average time. We denote the sorting function (this is the function mapping each list to its sorted version) by s, and we will work on the set of all total programs computing the function s. This set is denoted by $[s]'$.

7.1 Introductory notions

Definition 7.1 Given a countable total order $(\mathcal{A}, <)$, a *list* from $\mathcal{A}$ is a finite sequence of pairwise distinct (!) elements from $\mathcal{A}$. We use the restricted version of lists (that is lists consisting of pairwise distinct elements) in order to simplify the presentation. Define $Lists^{\mathcal{A}}$ to be the set of all lists obtained from $\mathcal{A}$. For any list $L \in Lists^{\mathcal{A}}$: $|L|$ denotes the *length* of the list L and we use $Lists^{\mathcal{A}}_n$ for the set of lists of length n.

A list is *sorted* when its elements (from left to right) are in increasing order with respect to to the ordering $<$ on $\mathcal{A}$. $\approx$ denotes the equivalence relation on $Lists^{\mathcal{A}}_n$ which identifies lists up to order isomorphism. $Lists^{\mathcal{A}}/\approx$ and $Lists^{\mathcal{A}}_n/\approx$ are denoted by $\mathcal{L}^{\mathcal{A}}$ and $\mathcal{L}^{\mathcal{A}}_n$ respectively.

Note that the cardinality of $\mathcal{L}^{\mathcal{A}}_n$ is $n!$.

In what follows we assume we have a fixed given total order $(\mathcal{A}, <)$ in mind and we will drop the superscript "$\mathcal{A}$" in $\mathcal{L}^{\mathcal{A}}$ and in $\mathcal{L}^{\mathcal{A}}_n$. This will simplify the notation without introducing ambiguities. Since we will always work with lists identified up to order isomorphism, we indicate the elements of $\mathcal{L}_n$ and $\mathcal{L}$ by $L, L', \ldots$, that is (with abuse of notation) we don't indicate the equivalence classes. Given a list $L \in \mathcal{L}_n$, we write $L = (L(1), \ldots, L(n))$.

Definition 7.2 Given a list L in $\mathcal{L}_n$, where $n \geq 2$: $L_1 = (L(1), \ldots, L(\lfloor \frac{n}{2} \rfloor))$ and $L_2 = (L(\lfloor \frac{n}{2} \rfloor + 1), \ldots, L(n))$.

Definition 7.3 A *sorting program* is a program which takes lists as inputs and returns the sorted version of these lists. A *comparison* made by a sorting program S between two different elements of a list L, say $L(i)$ and $L(j)$, is a determination (during the computation of $S(L)$) of their relative order, of the form "$L(i) < L(j)$" or "$L(j) < L(i)$". The *running time* of a sorting program S is defined to be:

$$T_S(L) = \begin{array}{l}\text{(the total number of comparisons made by } S \text{ on input } L\\ \text{during the computation of the sorted output } S(L)).\end{array}$$

The *average* running time (assuming uniform distribution on inputs) is defined by:

$$\overline{T_S}(n) = \frac{\sum_{|L|=n} T_S(L)}{n!}.$$

Note that this running time might be 0. For example for a program S which checks whether a list has length $|L| \leq 1$, and when this is the case, returns L as output, we have $\overline{T_S}(1) = 0$. We excluded 0-valued running times in order for the complexity distance not to result in a division by 0. However, as noted above, division only occurs through the second clause in the definition of the complexity distance (that is: "$T_P\,(n) > T_{P_2}(n)$", cf. section 4, definition 4.1), so inputs with zero time can be allowed as long as they don't fall under this clause. This will be the case under the (harmless) assumption that all sorting algorithms start with a length-check on the input L, and in case $|L| \leq 1$, return L. (Note that this assumption implies that all inputs L such that $|L| \leq 1$ have zero time, and that, as we work with comparison based sorting algorithms, these are the only inputs with zero time.) So we continue to work with the function space $\mathcal{C}_0$ in what follows. We give the usual definition of "asymptotic time".

Definition 7.4 $(\forall f, g \in \mathcal{C}_0)$:

1) $f \leq_O g$ iff $(\exists n_o)(\exists c > 0)(\forall n \geq n_o)\ f(n) \leq c \cdot g(n)$. We also use the (more standard) notation: "$f \in O(g)$" instead of "$f \leq_O g$".
2) $f \approx_O g$ iff $f \leq_O g$ and $g \leq_O f$. That is $\approx_O$ is the equivalence relation induced by the preorder $\leq_O$.

Definition 7.5 A program S has *optimal average asymptotic time* iff $[\overline{T}_S]$ is the minimum of the partial order $(\mathcal{C}_0/\approx_O, \leq_O)$.

Definition 7.6 A merging program is a program taking two sorted lists as inputs and returning the sorted list consisting of the union of their elements as output. Given a merging program *Merge*, a mergesort program (denoted by M) is defined by the following pseudo-code:

$$M(L) = \text{ if } |L| \leq 1, \text{ then return } L \text{ else return } Merge(ML_1, ML_2).$$

Definition 7.7 A *merge pair* is a pair of sorted lists.

$MPairs(m,n) = \{(L_1, L_2) \mid (L_1, L_2)$ is a merge pair and the lists L_1, L_2 are sublists of the unique sorted list of $\mathcal{L}_{m+n}$, of length m and n respectively$\}$.

Remark: The cardinality of $MPairs(m,n)$ is $\binom{m+n}{n}$.

Definition 7.8 $\overline{T}_{Merge}(m,n) = \frac{\sum_{(L_1,L_2)} T_{Merge}(L_1,L_2)}{\binom{n+m}{n}}$, where the sum ranges over $MPairs(m,n)$.

7.2 Recursion

Recall that $[s]'$ is the set of all comparison based sorting programs. Assume *Merge* is a merge program.

Definition 7.9 $M_2 : [s]' \to [s]'$ is defined by: for any program $S \in [s]'$,

$$M_2S(L) = [\text{ if } |L| \leq 1 \text{ then return } L \text{ else return } Merge(SL_1, SL_2)].$$

Note that with abuse of notation we do not indicate the dependence of M_2 on *Merge*.

Since the mergesort program will split each given list in two sublists, the identification up to order isomorphism will in general be broken as two lists which are distinct up to order isomorphism might have sublists with equivalent orders. We introduce the following notation to deal with this situation.

Definition 7.10 $\forall i \in \{1,2\}$: $List_i = \{L_i | L \in \mathcal{L}_n\}$ and $\mathcal{L}_i = List_i / \approx$.

Lemma 7.11 *For all $n \geq 2$, for all $L \in \mathcal{L}_n$, consider $[L_1] \in \mathcal{L}_1$ and $[L_2] \in \mathcal{L}_2$. The cardinality of $[L_1]$ is $\frac{n!}{\lfloor \frac{n}{2} \rfloor !}$ and the cardinality of $[L_2]$ is $\frac{n!}{(n - \lfloor \frac{n}{2} \rfloor)!}$.*

Proof. The equalities follow by an easy standard combinatorial argument. We give the proof for the cardinality of $[L_1]$. Given $L \in \mathcal{L}_n$, say $L = (L(1), \ldots, L(\lfloor \frac{n}{2} \rfloor), L(\lfloor \frac{n}{2} \rfloor + 1), \ldots, L(n))$. Note that since $L \in \mathcal{L}_n$, there are exactly n choices for $L(n)$. Once a choice is made for $L(n)$, there are $n-1$ choices for $L(n-1)$. Continuing this for each of the elements of L_2, one obtains that the possible choices for elements of L_2 are exactly $\frac{n!}{\lfloor \frac{n}{2} \rfloor !}$, that is each fixed ordering of L_1 can occur as a sublist of L in exactly $\frac{n!}{\lfloor \frac{n}{2} \rfloor !}$ many ways, where L_2 ranges over the possible orderings of the second part of the list L.$\square$

Corollary 7.12 *Let $n \geq 2$ and let $L_i^{'}$ be the sorted list of $\mathcal{L}_i$, where $i \in \{1,2\}$.*

$$\sum_{|L|=n} T_{MERGE}(L_1^{'}, L_2^{'}) = \sum_{(L_1^{'}, L_2^{'})} T_{MERGE}(L_1^{'}, L_2^{'}) \lfloor \frac{n}{2} \rfloor ! (n - \lfloor \frac{n}{2} \rfloor)!,$$

where $(L_1^{'}, L_2^{'})$ range over $MPairs(\lfloor \frac{n}{2} \rfloor, n - \lfloor \frac{n}{2} \rfloor)$.

Proof. This follows by an easy combinatorial argument. $\square$

Proposition 7.13 $(\forall S \in [s]')(\forall n \geq 2)$

$$\overline{T}_{M_2S}(n) = \overline{T}_S(\lfloor \frac{n}{2} \rfloor) + \overline{T}_S(n - \lfloor \frac{n}{2} \rfloor) + \overline{T}_{MERGE}(\lfloor \frac{n}{2} \rfloor, n - \lfloor \frac{n}{2} \rfloor).$$

Proof. $(\forall S \in [s]')(\forall n \geq 2)$

$$\overline{T}_{M_2S}(n) = \frac{\sum_{|L|=n} T_{M_2S}(L)}{n!} = \frac{\sum_{|L|=n}(T_S(L_1) + T_S(L_2) + T_{MERGE}(L_1', L_2'))}{n!}.$$

By lemma 7.11 and corollary 7.12 we obtain:

$$\begin{aligned}\overline{T}_{M_2S}(n) =& \frac{\sum_{|L\ |=\lfloor\frac{n}{2}\rfloor} T_S(L_1)\frac{n!}{\lfloor\frac{n}{2}\rfloor!}}{n!} + \frac{\sum_{|L_2|=(n-\lfloor\frac{n}{2}\rfloor)} T_S(L_2)\frac{n!}{(n-\lfloor\frac{n}{2}\rfloor)!}}{n!} \\ &+ \frac{\sum_{(L'\ ,L_2')} T_{MERGE}(L_1', L_2')\lfloor\frac{n}{2}\rfloor!(n-\lfloor\frac{n}{2}\rfloor)!}{n!} \\ =& \frac{\sum_{|L\ |=\lfloor\frac{n}{2}\rfloor} T_S(L_1)}{\lfloor\frac{n}{2}\rfloor!} + \frac{\sum_{|L_2|=(n-\lfloor\frac{n}{2}\rfloor)} T_S(L_2)}{(n-\lfloor\frac{n}{2}\rfloor)!} \\ &+ \frac{\sum_{(L'\ ,L_2')} T_{MERGE}(L_1', L_2')}{\frac{n!}{\lfloor\frac{n}{2}\rfloor!(n-\lfloor\frac{n}{2}\rfloor)!}} \\ =& \overline{T}_S(\lfloor\frac{n}{2}\rfloor) + \overline{T}_S(n-\lfloor\frac{n}{2}\rfloor) + \overline{T}_{MERGE}(\lfloor\frac{n}{2}\rfloor, n-\lfloor\frac{n}{2}\rfloor).\end{aligned}$$

□

Recall the following well known result:

Lemma 7.14 *If f is a monotone increasing function, then* $(\forall n \in \{2^k \mid k \geq 0\}) f(n) \leq nlog_2(n) \;\Rightarrow\; (\exists c > 0)(\forall n \geq 1)\ f(n) \leq cnlog_2(n))$.

As we work with comparison based sorting algorithms we can without loss of generality make the assumption that the complexity functions of these programs are monotone increasing. So in what follows we will assume the list lengths to be a power of 2. In particular Proposition 7.13 reduces to:

$$(\forall n \geq 2)(\forall S \in [s]')\ \overline{T}_{M_2S}(n) = 2\overline{T}_S(\frac{n}{2}) + \overline{T}_{MERGE}(\frac{n}{2}, \frac{n}{2}).$$

In order to obtain the application, we need to make a careful analysis of the average time required to merge, that is we will determine the term $\overline{T}_{MERGE}(\frac{n}{2}, \frac{n}{2})$ of the above recurrence equation.

7.3 Optimal Merging

The study of the minimum average number of comparisons necessary to merge "m things with n" is an open problem stated in [7] (cf. the research problem listed as exercise 22, section 5.3.3). In [7] the worst-case analysis of merging is made via the function $M(m,n)$ representing the minimum number of comparisons sufficient to merge m things with n. A lower bound and an upper bound for $M(m,n)$ is obtained, and in the special case where $m = n$, a precise value for $M(m,m)$ is obtained: $M(m,m) = 2m - 1$. The proof shows that no (comparison based) algorithm can have better worst-case time, and secondly notes that the standard merge algorithm (cf. [1],[7]) has worst-case time $T(m,m) = 2m - 1$. Knuth notes that the right hand side expression of this equality corresponds to the upper bound obtained for $M(m,n)$ when $m = n$ (cf. [7]), and argues that the lower bound is therefore "at fault". We show that the lower bound actually represents the minimum *average* number of comparisons necessary to merge m things with n.

We solve the problem on the minimum average number of comparisons required to merge, by determining a lower bound and by solving the case for $m = n$.

Proposition 7.15 (Lower bound on average mergetime) *For any comparison based merge algorithm M, $\overline{T}_M(m,n) \geq \lceil log_2\binom{n+m}{m}\rceil$.*

Proof. Follows by a standard comparison tree analysis. □

Corollary 7.16 *For any comparison based merge algorithm M, $\overline{T}_M(n,n) \geq n$.*

Proof. By proposition 7.16, $\overline{T}_M(n,n) \geq \lceil log_2\binom{2n}{n}\rceil$. Note that

$$\lceil log_2\binom{2n}{n}\rceil = \lceil log_2\frac{(2n)!}{(n!)^2}\rceil = \lceil log_2\frac{2n(2n-1)\ldots(n+1)}{n!}\rceil$$
$$\geq \lceil log_2\frac{(2n)(2n-1)\ldots(2n-(n-1))}{n(n-1)\ldots 1}\rceil.$$

Since $(\forall y \geq 0)\ 2x - y \geq 2(x-y)$, we obtain

$$\overline{T}_M(n,n) \geq \lceil log_2\frac{(2n)(2(n-1))\ldots(2(n-(n-1)))}{n(n-1)\ldots 1}\rceil = \lceil log_2 2^n\rceil = n. \quad \square$$

In the worst-case analysis performed in [7] the upper bound is shown to be "exact" in the sense that: $M(m,m) = 2m-1$. We obtain a similar result for the lower bound for average merging time via the introduction of an oracle $\mathcal{O}$.

Definition 7.17 A *consecutive* sublist of a list $L = (L(1),\ldots,L(n))$, is a sublist $(L(i_1),\ldots,L(i_k))$ of L such that for each $j : 1\ldots k-1 : L(i_j+1)$ is the successor of $L(i_j)$ with respect to the total order on list elements.

There is no guarantee that a merge algorithm exists with optimal average time $\overline{T}(n,n) = n$. However, an "ideal" merge program I can be defined, using an oracle $\mathcal{O}$ which for every merge pair (L_1, L_2) yields the positions of the heads of the consecutive sublists in both lists without cost in comparison time.

Recall that the *standard merge algorithm* is defined by the following pseudocode (where L_1 and L_2 represent sorted lists, head(L) represents the head of the list L, and tail(L) is the list obtained by removing the head from L):

```
Merge($L_1$,$L_2$) =
[Let $L' = \emptyset$
While $L_1 \neq \emptyset$ and $L_2 \neq \emptyset$ repeat:
If head($L_1$)<head($L_2$)
  then append($L'$,head($L_1$)) and let $L_1$ = tail($L_1$)
  else append($L'$,head($L_2$)) and let $L_2$ = tail($L_2$)
If $L_1 = \emptyset$
```

```
    then append($L'$,$L_2$)
    else append($L'$,$L_1$)]
```

Given the oracle $\mathcal{O}$, we define the following operations. For every sorted list L, let $H(L)$ denote the list consisting of the heads of the consecutive sublists of L in left to right order, obtained via the oracle $\mathcal{O}$. Note that this list is still sorted. For any element a of $H(L)$, $\overline{a}$ denotes the consecutive list in L starting with a. Note that this operation is well defined by our assumption that list elements are different from one another. We extend the operation to lists $L = (L(1), \ldots, L(n))$ by defining $\overline{L}$ to be the concatenation of the lists $\overline{L(1)}, \ldots, \overline{L(n)}$ (where we assume that the list elements of L are heads of consecutive sublists of a given number of disjoint sorted lists).

We define the *ideal merge program I* via the following pseudo-code (where *Merge* is the standard merge algorithm):

```
$I(L_1, L_2) =$
[Let $L' = Merge(H(L_1), H(L_2))$, Return $\overline{L'}$]
```

The *ideal mergesort algorithm* defined via the ideal merge algorithm I, is denoted by $\mathcal{M}$.

The following theorem gives the exact value of the average time the ideal merge program spends on merging lists of identical length.

Theorem 7.18 $(\forall n \geq 1)$ $\overline{T_I}(n,n) = n$.

Before presenting the proof, we state the following corollary. Given the oracle $\mathcal{O}$, define

$\overline{M}_{\mathcal{O}}(m,n) =$ the minimum average time *sufficient* to merge m things with n,

The following corollary gives the exact value of $\overline{M}_{\mathcal{O}}(n,n)$.

Corollary 7.19 $\overline{M}_{\mathcal{O}}(n,n) = n$.

Proof. Immediate from Corollary 7.16 and Theorem 7.18. □

In order to prove Theorem 7.18, we introduce an encoding of merge pairs, and a lemma dealing with the number of "alternations" of merge pair encodings.

Recall that we assume that lists have a length n which is a power of 2.

Note that when $n = 2k$, $\overline{T}_I(k,k) = \frac{\sum_{(L_1,L_2)} T_I(L_1,L_2)}{\binom{n}{k}}$, where the sum ranges over $MPairs(k,k)$.

The number of non-order-isomorphic merge pairs obtainable from elements of the sorted list of length n, say L', is $\binom{n}{k}$. This number corresponds to the possible number of ways a sorted list L_1' can be obtained from the sorted list $L' = (1, \ldots, n)$.

In order to simplify the analysis, we encode the merge pairs via a "binary encoding", which associates a binary list with each given merge pair. Each choice of L_1' from L' can be encoded via a binary list l of length n, where an occurrence of 1 indicates a choice of an element of L'_1, and an occurrence of 0 indicates a choice of an element of L'_2.

For example consider the sorted list of length 6: $L' = (1,2,3,4,5,6)$: The pair consisting of L'_1 = (1,3,6) and L'_2 = (2,4,5) is encoded via the binary list: l = (1,0,1,0,0,1).

Notation: $l_{1,2}$ denotes the binary encoding of the pair (L'_1, L'_2). An *alternation* in a binary list l is a change from 1 to 0 or from 0 to 1. Given a binary list l, let $\mathcal{A}(l)$ denote the number of alternations in l. For example the binary encoding $l_{1,2}$ of the pair (L'_1, L'_2) given above has 4 alternations. Note: $T_I(L'_1, L'_2) = \mathcal{A}(l_{1,2})$.

In order to prove the theorem, we need the following lemmas.

Lemma 7.20 *Let* $k = |L'_1| = |L'_2| = \frac{n}{2}$. *Then*

$$\sum_{(L',L'_2)} T_I(L'_1, L'_2) = 2[\sum_{i=0}^{k-1} \binom{k-1}{i}^2 (2i+1) + \sum_{i=0}^{k-1} \binom{k-1}{i}\binom{k-1}{i-1}(2i)].$$

Proof. From the note made above, it suffices to show:

$$\sum_{(L',L'_2)} \mathcal{A}(l_{1,2}) = 2[\sum_{i=0}^{k-1} \binom{k-1}{i}^2 (2i+1) + \sum_{i=0}^{k-1} \binom{k-1}{i}\binom{k-1}{i-1}(2i)].$$

To count the number of alternations it suffices to consider only the binary lists starting with 1, and to multiply the result by 2 (remark that any binary list l has the same number of alternations as its "negative" version, that is the version obtained by replacing each element i of l by $1-i$).

Given $i \in \{1,0\}$: a binary list l of which all the elements are equal to i is referred to as an "i-list". Consider l_1, the 1-list of length k. Let $n = 2k$.

The possible alternations in a binary list l of length n, which has k occurrences of the element 1 and k occurrences of the element 0, are determined by the possible insertions of 0-lists among the elements of l_1. There are $k-1$ posssibilities for these insertions, and one extra possibility corresponding to the appending of a 0-list to the list l_1. Since insertions of 0-lists each contribute 2 alternations and an appended 0-list only contributes one alternation, we consider two cases:

1) $\mathcal{A}(l_{1,2})$ is odd (that is l ends in a 0-list): We have $\binom{k-1}{i}$ ways to obtain i insertions, where $i : 1, \ldots, k-1$. The insertion of i 0-lists (in the i places chosen between the elements of the 1-list l_1) and the appending of the 0-list to l_1 can happen in different ways, depending on how the 0-list is split into the $i+1$ sublists used in this proces. There are i divisions to be chosen in order to split a 0-list into $i+1$ sublists, that is there are $\binom{k-1}{i}$ ways to split the 0-list l_2 in $i+1$ sublists.

So when $\mathcal{A}(l_{1,2})$ is odd:

$$\mathcal{A}(l_{1,2}) = \sum_{i=0}^{k-1} \binom{k-1}{i}\binom{k-1}{i}(2i+1) = \sum_{i=0}^{k-1} \binom{k-1}{i}^2 (2i+1).$$

2) $\mathcal{A}(l_{1,2})$ is even (that is l ends in a 1-list):

$$\mathcal{A}(l_{1,2}) = \sum_{i=1}^{k-1} \binom{k-1}{i}\binom{k-1}{i-1}(2i).$$

The result is obtained by a similar argument. Note that the sum starts from one on, since as we have no 0-list appended to l_1 in this case, we must at least have one insertion of a 0-list in l_1. Note that there is a spot to insert this 0-list, as the length of l_1 must be at least 2.

Recall that we need to multiply the result by 2 in order to account for the negative versions of the binary lists, that is the lists starting with 0. $\square$

Lemma 7.21 *For $n = 2k$, $k \geq 1$,*

$$\overline{T}_I(n,n) = \frac{2k\left\{\sum_{i=0}^{k-1}\binom{k-1}{i}^2 + \sum_{i=1}^{k-1}\binom{k-1}{i}\binom{k-1}{i-1}\right\}}{\binom{n}{k}}.$$

Proof. Note that

$$2\sum_{i=1}^{k-1}\binom{k-1}{i}\binom{k-1}{i-1}(2i) =$$
$$\left\{\binom{k-1}{1}\binom{k-1}{0}2 + \ldots + \binom{k-1}{k-1}\binom{k-1}{k-2}(2k-2)\right\}$$
$$+\left\{\binom{k-1}{k-1}\binom{k-1}{k-2}(2k-2) + \ldots + \binom{k-1}{1}\binom{k-1}{0}2\right\}.$$

Adding up the two sums term by term (in left to right order) we obtain $2k\sum_{i=1}^{k-1}\binom{k-1}{i}\binom{k-1}{i-1}$.

Similarly,

$$2\sum_{i=0}^{k-1}\binom{k-1}{i}^2(2i+1) = \left\{\binom{k-1}{0}^2 1 + \ldots + \binom{k-1}{k-1}^2(2k-1)\right\}$$
$$+\left\{\binom{k-1}{k-1}^2(2k-1), + \ldots + \binom{k-1}{0}^2 1\right\}$$
$$= 2k\sum_{i=0}^{k-1}\binom{k-1}{i}^2.$$

So we have $\sum_{(L_1',L_2')} T_I(L_1',L_2') = 2k\left\{\sum_{i=0}^{k-1}\binom{k-1}{i}^2 + \sum_{i=1}^{k-1}\binom{k-1}{i}\binom{k-1}{i-1}\right\}$. That is for $n = 2k : \overline{T}_I(n,n) = \frac{2k\left\{\sum_{i=0}^{k-1}\binom{k-1}{i}^2 + \sum_{i=1}^{k-1}\binom{k-1}{i}\binom{k-1}{i-1}\right\}}{\binom{n}{k}}$ $\square$

Lemma 7.22 *For k, l such that $k \leq 2l$,*

$$\binom{2l}{k} = \sum_{i=max\{0,k-l\}}^{l}\binom{l}{i}\binom{l}{l-(i-(max\{0,k-l\}))}.$$

Proof. We distinguish two cases: $k \leq l$ and $k > l$. The result follows for the first case by the remark that the number of ways to choose k objects among

$2l$ given objects, is the sum of the number of ways to choose i objects among the first l, multiplied by the number of ways to choose the remaining $k-i$ objects among the l remaining given objects.

The case where $l < k \leq 2l$ follows by a similar argument, noting that one is forced to pick at least $k-l$ objects among the first l. □

Lemma 7.23 $(\forall k \geq 1)$

$$2\left\{\binom{2k-2}{k-1} + \binom{2k-2}{k}\right\} = \binom{2k}{k}.$$

Proof. Since $k \geq 1$ we have $2k \geq 2$, so there are at least 2 elements to choose from. Note that k elements can be chosen among $2k$ given elements by
1) selecting two elements among the last two of the $2k$ given ones, and by picking the $k-2$ remaining elements among the first $2k-2$ given elements: $\binom{2k-2}{k-2} = \binom{2k-2}{k}$ possibilities;
2) selecting one element in one of the last two positions among the $2k$ given ones, and the $k-1$ remaining ones among the first $2k-2$ given elements: $2\binom{2k-2}{k-1}$ possibilities;
3) selecting all k elements among the first $2k-2$ given elements: $\binom{2k-2}{k}$ possibilities. □

Finally we are ready to show the theorem, that is: $(\forall k \geq 1)\ \overline{T}_{\mathcal{M}}(k,k) = k$, where $k = \frac{n}{2}$.

Proof of Theorem 7.18. By Lemma 7.22 we obtain

(1) $\sum_{i=0}^{k-1} \binom{k-1}{i}^2 = \sum_{i=0}^{k-1} \binom{k-1}{i}\binom{k-1}{(k-1)-i} = \binom{2k-2}{k-1}$ and

(2) $\sum_{i=1}^{k-1} \binom{k-1}{i}\binom{k-1}{i-1} = \sum_{i=1}^{k-1} \binom{k-1}{i}\binom{k-1}{(k-1)-(i-1)} = \binom{2k-2}{k}$.

so by Lemma 7.21 $\overline{T}_I(k,k) = \frac{\sum_{(L_1,L_2)} T_I(L_1,L_2)}{\binom{n}{k}} = \frac{2k\{\binom{2k-2}{k-1}+\binom{2k-2}{k}\}}{\binom{n}{k}}$, and hence $\overline{T}_I(k,k) = k \Leftrightarrow \frac{2k\{\binom{2k-2}{k-1}+\binom{2k-2}{k}\}}{\binom{2k}{k}} = k \Leftrightarrow 2\left\{\binom{2k-2}{k-1} + \binom{2k-2}{k}\right\} = \binom{2k}{k}$. The last equality holds by Lemma 7.23, and thus $\overline{T}_I(k,k) = k$. □

7.4 Optimal asymptotic average time of mergesort

Our goal is to show an optimization result for mergesort programs. For these programs the (average) time is monotone in the (average) time of the merge algorithm on which they are based (Proposition 7.13). Since linear time merge programs exist (for example the standard merge program), we need only consider linear time merge programs in the optimization analysis. The following lemma provides a justification for the fact that in this analysis the ideal merge program is representative for the class of the linear time merge programs.

Lemma 7.24 *$O(\overline{T}_{\mathcal{M}}) = O(\overline{T}_M)$, for any mergesort program, based on a merge program "Merge" such that $\overline{T}_{Merge} \in O(n)$.*

Proof. By Corollary 7.16 we know that for any merge program "Merge":

$\overline{T}_{Merge}(n,n) \geq n$. So since $\overline{T}_I(n,n) = n$, we have: $\overline{T}_I \leq \overline{T}_{Merge}$ and thus $\overline{T}_{\mathcal{M}} \leq \overline{T}_M$.

Conversely, since $\overline{T}_{Merge} \in O(n)$, $(\exists c > 0)(\forall n \geq 1)\ \overline{T}_{Merge}(n) \leq cn$. So $(\forall n \geq 2)\overline{T}_M(n) = 2\overline{T}_M(\frac{n}{2}) + \overline{T}_{Merge}(\frac{n}{2},\frac{n}{2}) \leq 2\overline{T}_M(\frac{n}{2}) + c\frac{n}{2}$. We also have $(\forall n \geq 2)\ \overline{T}_{\mathcal{M}}(n) = 2\overline{T}_{\mathcal{M}}(\frac{n}{2}) + \frac{n}{2}$, and thus $(\forall n)\ \overline{T}_M(n) \leq c\overline{T}_{\mathcal{M}}(n)$ (easy proof by induction). □

Note that by Theorem 7.18, we obtain that the equation given under Proposition 7.13 reduces to

$$(\forall n \geq 2)(\forall S \in [s]')\ \overline{T}_{M_2 S}(n) = 2\overline{T}_S(\frac{n}{2}) + \frac{n}{2}.$$

The functional Φ_2 corresponding to the recurrence equation which determines $\overline{T}_{\mathcal{M}}$ therefor is defined by:

$$\Phi_2 \colon \mathcal{C}_0|b \to \mathcal{C}_0|b, \quad \text{where } \Phi_2 = \lambda f \lambda n.\ \text{if } n = 1 \text{ then } 0 \text{ else } 2f(\frac{n}{2}) + \frac{n}{2}.$$

Lemma 7.25 *Φ_2 is an improver with respect to the function $g(n) = c'nlog_2(n)$ $\Leftrightarrow c' \geq \frac{1}{2}$*

Proof. Since Φ_2 is monotone on $\mathcal{C}_0$, it suffices to show that $\Phi_2 g \leq g \Leftrightarrow c' \geq \frac{1}{2}$. Note that when $n = 1$, we have $(\Phi_2 g)(n) = 0 = g(n)$ and when $n > 1$ $(\Phi_2 g)(n) = 2(c'\frac{n}{2}log_2\frac{n}{2}) + \frac{n}{2}$, so

$$\begin{aligned}(\Phi_2 g)(n) \leq g(n) &\Leftrightarrow 2(c'\frac{n}{2}log_2\frac{n}{2}) + \frac{n}{2} \leq c'nlog_2(n)\\ &\Leftrightarrow 2c'\frac{n}{2}(log_2(n) - 1) + \frac{n}{2} \leq c'nlog_2 n\\ &\Leftrightarrow c'n - \frac{n}{2} \geq 0\\ &\Leftrightarrow c' \geq \frac{1}{2}.\end{aligned}$$

□

So by Proposition 6.3 and Lemma 7.25 we have $\overline{T}_{\mathcal{M}} \leq_d \frac{1}{2}nlog_2 n$ or $\mathcal{M} \in O(nlog_2 n)$ and thus by Lemma 7.24 we obtain that every mergesort program based on a linear average time merge algorithm has optimal asymptotic average time.

8 Conclusion

A complexity distance on programs has been defined, providing a new means to perform complexity analyis. Its properties guarantee that the spaces equipped with this distance, that is the complexity (distance) spaces, have a sequential S-completion. This implies that these spaces can be studied within the topological framework of Denotational Semantics (suggesting the possibility that this framework might allow for a future combination of Denotational Semantics and Complexity Analysis into an intensional semantics). A version of the Banach theorem has been shown for this particular sequential S-completion. The Divide & Conquer algorithms have been shown to induce contraction maps on this completion. The applicability of the theory has been illustrated by the complexity analysis of a particular Divide & Conquer algorithm. The distance thus has interesting properties both from a theoretical and a practical point of view.

References

[1] V. Aho, J. Hopcroft, J. Ullman, Datastructures and algorithms 1987, Addison-Wesley.

[2] B. Bjerner, Time complexity of programs in type theory, 1989, University of Goteborg.

[3] M. Davis, E. Weyuker, Computability, complexity and languages, 1983, N.Y. Academic Press.

[4] A. Di Concilio, Spazi quasimetrici e topologie ad essi associate, Accademia di Scienze Fisiche e Matematiche, Lettere ed Arti in Napoli, Serie 4 - Vol. XXXVIII, 1971.

[5] J. Dugundji, Topology, 1966, Allyn and Bacon, Inc., Boston.

[6] P. Fletcher, W. Lindgren, Quasi-uniform spaces, 1982, Marcel Dekker, Inc., NY.

[7] D. Knuth, The art of computer programming vol.3, 1973, Addison-Wesley.

[8] H. P. Kunzi, Nonsymmetric Topology, Proceedings Szekszard Conference, 1993.

[9] H. P. Kunzi, Complete quasi-pseudo-metric spaces, Acta Math. Hung. 59 ,1992.

[10] H.P. Kunzi, V. Vajner, Weighted quasi-metrics, Proc. Summer Conf. Queens College, Gen. Top. Appl., 1993, Proc. NY Acad. Sci.

[11] Ming Li, P. Vitanyi, An introduction to Kolmogorov Complexity and its applications, 1993, Springer Verlag.

[12] L. Nachbin, Topology and order, New York Mathematical Studies (vol. 4), 1965, Princeton, N.J.

[13] S.G. Matthews, Partial metric spaces, research report RR212, 1992, University of Warwick.

[14] S.G. Matthews, The topology of partial metric spaces, research report RR222, 1992, University of Warwick.

[15] M. Smyth, Completeness of quasi-uniform and syntopological spaces, manuscript, Imperial College.

[16] M. Smyth, Quasi-uniformities: Reconciling domains with metric spaces, LNCS 298, 1987, Springer Verlag.

[17] M. Smyth, Totally bounded spaces and compact ordered spaces as domains of computation, Topology and Category Theory in Computer Science, p. 207-229, Oxford, 1991, Oxford University Press.

[18] P. Sünderhauf, The Smyth completion of a quasi-uniform space, preprint 1427, 1991, Technische Hochschule Darmstadt.

[19] P. Sünderhauf, Quasi-uniform completeness in terms of Cauchy nets, preprint 1529, 1992, Technische Hochschule Darmstadt.