Tight Bounds on Online Checkpointing Algorithms

Achiya Bar-On
Department of Mathematics, Bar-Ilan University, Ramat Gan, Israel
abo1000@gmail.com

Itai Dinur[1]
Computer Science Department, Ben-Gurion University, Beer Sheva, Israel
dinuri@cs.bgu.ac.il

Orr Dunkelman
Computer Science Department, University of Haifa, Haifa, Israel
orrd@cs.haifa.ac.il
  https://orcid.org/0000-0001-5799-2635

Rani Hod
Department of Mathematics, Bar-Ilan University, Ramat Gan, Israel
rani.hod@math.biu.ac.il

Nathan Keller[2]
Department of Mathematics, Bar-Ilan University, Ramat Gan, Israel
nkeller@math.biu.ac.il

Eyal Ronen
Computer Science Department, The Weizmann Institute, Rehovot, Israel
eyal.ronen@weizmann.ac.il

Adi Shamir
Computer Science Department, The Weizmann Institute, Rehovot, Israel
adi.shamir@weizmann.ac.il

Abstract

The problem of online checkpointing is a classical problem with numerous applications which had been studied in various forms for almost 50 years. In the simplest version of this problem, a user has to maintain k memorized checkpoints during a long computation, where the only allowed operation is to move one of the checkpoints from its old time to the current time, and his goal is to keep the checkpoints as evenly spread out as possible at all times.

At ICALP'13 Bringmann et al. studied this problem as a special case of an online/offline optimization problem in which the deviation from uniformity is measured by the natural discrepancy metric of the worst case ratio between real and ideal segment lengths. They showed this discrepancy is smaller than $1.59 - o(1)$ for all k, and smaller than $\ln 4 - o(1) \approx 1.39$ for the sparse subset of k's which are powers of 2. In addition, they obtained upper bounds on the achievable discrepancy for some small values of k.

In this paper we solve the main problems left open in the ICALP'13 paper by proving that $\ln 4$ is a tight upper and lower bound on the asymptotic discrepancy for all large k, and by providing tight upper and lower bounds (in the form of provably optimal checkpointing algorithms, some of which are in fact better than those of Bringmann et al.) for all the small values of $k \leq 10$.

2012 ACM Subject Classification Theory of computation → Online algorithms

1   Introduction and Notation

Most programs perform some irreversible operations, and thus they can only be run in a forward direction. However, in many cases we would like to roll back a computation to an earlier point in time. When the computation is short, we can just rerun the computation from the beginning, but when the computation requires many days, a better strategy is to memorize several copies of the full state of the computation at various times. These memorized states (called *checkpoints*) make it possible to roll the computation back from time T to any earlier time $T' < T$ by restarting the computation from the last available checkpoint which was memorized before T'. This checkpointing technique is extremely useful in many real life applications: For example, when we want to interactively debug a new program we may want to randomly access earlier points in the execution in order to find the source of a problem; in fault tolerant computer systems we may want to undo the effect of faulty hardware; and during lengthy simulations of physical systems we may want to explore the effect of changing some parameter such as the temperature at some earlier point in time without rerunning the simulation from the beginning.

In principle, we can try to memorize the full state of the computation after each step, but for long computations this requires an unrealistic amount of memory. Instead, we assume that we have some bounded amount of memory which suffices to keep k checkpoints. At time T, these checkpoints are spread within the time interval $[0, T]$, dividing it into $k + 1$ subintervals between consecutive checkpoints (where the endpoints 0 and T can be viewed as virtual checkpoints which require no additional memory). As T increases, the last subinterval gets longer, and at some point we may want to relocate one of the old checkpoints by reusing its memory to store the current state of the computation. A checkpointing algorithm can thus be viewed as an infinite pebbling game in which we place k pebbles on the positive side of the time axis, and then repeatedly perform update operations which move one of the pebbles to the right of all the other pebbles.

The first paper dealing with this problem seems to be "Rollback and Recovery Strategies for Computer Programs" [5], published in 1972, while the first paper which tried to solve it optimally was "On the Optimum Checkpoint Interval" [7], published in 1979. Over the years, dozens of academic research papers were published in this area, most notably [8] in 1984, [3] in 1994, and [1, 4] in 2013. However, many of these papers either dealt with concrete applications of the problem in other areas (especially in distributed computing where the notion of a timeline is different), or used other optimization criteria (which make their optimal solutions incomparable with ours). The mathematical problem we are dealing with in this paper was mentioned in [1] and studied in [4] which was published at ICALP'13, and we closely follow their model and notation.

At any time T, we define a *snapshot* as the ordered sequence of current checkpoint locations $S = (T_1, \ldots, T_k)$. Within each snapshot, we refer to the checkpoints by their *freshness index* p, where checkpoint 1 stores the oldest state and checkpoint k stores the newest state. Starting from an initial snapshot $S_k = (t_1, t_2, \ldots, t_k)$, we define for every
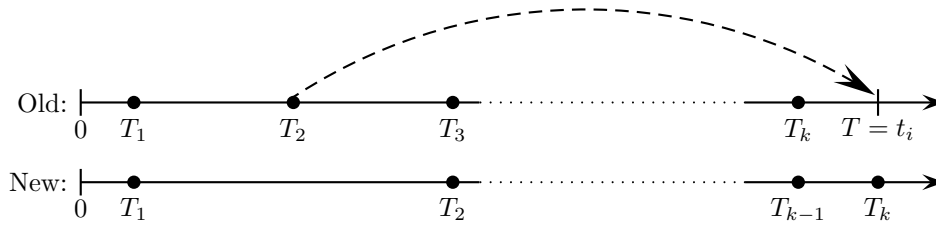
Figure 1 Transition from old to new snapshot for the update action $(t_i, 2)$.

$i \geq k + 1$ the i-th *update action* as a pair (t_i, p_i) in which p_i is the freshness index of the checkpoint whose memory we want to reuse by moving it to time t_i. A typical example of how one snapshot is transformed into another snapshot by an update operation is described in Fig. 1. The effect of the i-th update action is to unify the two consecutive subintervals which were separated by the p_i-th oldest active checkpoint at time $T = t_i$, and to create a new subinterval which ends at t_i. Note that with this notation, each update action affects multiple freshness indices within the snapshot; in particular, the freshness index of active checkpoint T_j for $1 \leq j < p_i$ is left unchanged, and is decreased by one for $p_i < j \leq k$. To demonstrate this point, consider a sequence of updates in which $p_i = 1$ for all $i \geq k + 1$: it updates the k memory locations in a round robin way since it always updates the oldest active checkpoint by overwriting it with the newest checkpoint, shifting all freshness indices by one. On the other hand, a sequence of updates in which $p_i = k$ for all $i \geq k + 1$ keeps updating the same memory location, pushing its associated checkpoint further and further to the right, with no change to the other checkpoints.

In this model, the time complexity of rolling back a computation from time T to time T' is assumed to be proportional to the distance between T' and the last checkpoint that precedes T' in the snapshot at time T, and thus its worst case happens when we decide to roll back to just before the end of the longest subinterval. A *checkpointing algorithm* (t, p) consists of a monotonically increasing and unbounded sequence of update times $t = \{t_i\}_{i=1}^{\infty}$ and a pattern sequence $p = \{p_i\}_{i=k+1}^{\infty}$, forming an initial snapshot and an infinite sequence of update actions; its goal is to make the length of this longest subinterval as short as possible. Clearly, no checkpointing algorithm can make this length shorter than the subinterval length in a perfectly uniform partition of $[0, T]$, which is $T/(k + 1)$. We say that a snapshot $S = (T_1, \ldots, T_k)$ of a k-checkpoint algorithm $\mathrm{ALG} = (t, p)$ is *q-compliant* at time T if the $k + 1$ subintervals defined by S satisfy $T_j - T_{j-1} \leq qT/(k + 1)$ for $j = 1, \ldots, k+1$,[3] and that ALG is *q-efficient* if its snapshots are q-compliant at all times $T \geq t_k$. Finally, the *efficiency* of a checkpointing algorithm is defined as the smallest q for which it is q-efficient.

Notice that the problem of efficient checkpointing can be viewed as a special case of an online/offline optimization problem: If we knew in advance the time T at which we would like to roll back the computation, we could make each subinterval as small as $T/(k + 1)$. However, in the online version of the problem, we do not know T in advance, and thus we have to position the checkpoints so that they will be roughly equally spaced at all times. The efficiency of the solution is the ratio between what we can achieve in the online and offline cases, respectively, and the goal of the *online checkpointing problem* is to find the smallest possible efficiency q_k achievable by the best k-checkpoint algorithm for any given k.

---

[3] We write $T_0 = 0$ and $T_{k+1} = T$ for convenience.

Clearly, $q_k \geq 1$ for all $k \geq 2$, and cannot be too close to 1 since any snapshot in which all subintervals have roughly the same length will be transformed by the next update operation to a snapshot in which one of the subintervals will be the union of two previous subintervals, and thus will be about twice as long as the other subintervals.[4] On the other hand, there is a very simple subinterval doubling algorithm from [1, Section 3.1] which is 2-efficient: Assuming WLOG that k is even, the algorithm starts with the snapshot $(1, 2, 3, \ldots, k)$, and performs the sequence of update actions $(k+2, 1), (k+4, 2), \ldots, (2k, k/2)$, yielding the snapshot $(2, 4, 6, \ldots, 2k)$. Since this snapshot is the same as the original snapshot up to a scaling factor of 2, we can continue with update actions $(2k+4, 1), (2k+8, 2), \ldots, (4k, k/2)$ and so on. This is a *cyclic* algorithm, repeating the same sequence of freshness indices again and again but with times which form a geometric progression. As in each snapshot there are only two possible lengths for the subintervals of the form x and $2x$, all the snapshots in this algorithm are 2-compliant, and thus the algorithm is 2-efficient.

The best strategy for keeping the checkpoints as uniform as possible at all times is thus to keep in each snapshot a variety of subinterval lengths, so that the algorithm will always be able to join two relatively short adjacent subintervals into a single subinterval which is not too long. This can be viewed as a generalization of the algorithm that creates Fibonacci numbers: Whereas the standard algorithm is always adding the last two numbers and placing their sum on the right, in our case we can add any two consecutive numbers in the sequence, replacing them by their sum and adding any number we want on the right. Analyzing this problem is surprisingly difficult, and so far there had been no tight bounds on the best possible efficiencies q_k of online checkpointing algorithms in this model.

The main results in [4] are two online checkpointing algorithms whose asymptotic efficiencies are $\ln 4 + o(1) \approx 1.39$ for the sparse subset of k's which are powers of 2, and 1.59 for general k. In addition, they proved in their model the first nontrivial asymptotic lower bound of $2 - \ln 2 - o(1) \approx 1.30$. However, since the upper and lower bounds did not match, it was not clear whether the checkpointing algorithms they proposed were asymptotically optimal. For small values of $k < 60$ they presented concrete checkpointing algorithms whose efficiencies were all below 1.55, but again it was not clear whether they were optimal.

In this paper we solve the main open problems related to the mathematical formulation of the problem which was defined and studied in [4]. In particular, we develop a new checkpointing algorithm with an asymptotic efficiency of $\ln 4$ for all values of k, and prove its optimality by providing a matching asymptotic lower bound. For all the small values of $k < 10$ we develop optimal checkpointing algorithms by proving tight upper and lower bounds on the achievable efficiency for these k's. This analysis enables us to show that for some values of k (such as $k = 8$), the algorithms presented in [4] are in fact suboptimal.

The rest of this paper is organized as follows. In Section 2 we go over basic observations about checkpointing algorithms (some from [4], some new). In Section 3 we focus on moderately small values of k and provide optimal algorithms for $k \leq 10$. In Section 4 we construct a recursive algorithm of asymptotically optimal efficiency $\ln 4 + o(1)$. In Section 5 we prove a matching asymptotic lower bound of $\ln 4 - o(1)$. In Section 6 we provide concluding remarks.

Most proofs were omitted from this extended abstract due to space constraints.

---

[4] Actually $q_k \geq (k+1)/k$ since subinterval $k+1$ has zero length upon updating, as noted in [1, Theorem 3].

2   Basic Observations

By definition, a k-checkpoint $\text{ALG} = (t, p)$ is q-efficient if and only if its snapshots at all times $T \geq t_k$ are q-compliant. However, as noted in [1, Lemma 2] (and also [4, Lemma 1]), it suffices to verify compliance only at the discrete times $T \in \{t_i\}_{i=k}^{\infty}$. It makes sense thus to only consider "standard" snapshots S_i taken at time t_i for $i \geq k$. Moreover, as shown in [4, Lemma 2], besides compliance of the initial snapshot S_k, it suffices to verify compliance of just two subintervals of S_i for every $i > k$ — subinterval k, that ends in the new checkpoint t_i, and subinterval p_i, created by merging two consecutive subintervals.

The following two observations about the sequence $p = (p_i)_{i=k+1}^{\infty}$ were mentioned in [4, Section 6] without proof.

▶ **Fact 1.** *Without loss of generality we can assume a k-checkpoint algorithm updates the least recent checkpoint infinitely often (i.e., $\liminf\limits_{i \to \infty} p_i = 1$).*

▶ Remark 2. An important consequence of Fact 1 is that we can essentially ignore the compliance of the initial snapshot S_k by *rebasing*, i.e., running the algorithm until all checkpoints present in S_k are overwritten and treating the then-current snapshot as the new initial $(t_1, \ldots, t_k)$.

▶ **Fact 3.** *Without loss of generality we can assume a k-checkpoint algorithm never updates the most recent checkpoint (i.e., $p_i < k$ for all $i \geq 1$).*

▶ Remark 4. Fact 3 means that the two last checkpoints in snapshot S_i are t_{i-1} and t_i, and thus subinterval k is q-compliant if and only if $t_{i-1} - t_i \leq qt_i / (k+1)$, that is, $t_i \leq G \cdot t_{i-1}$, where $G = G(q) := (k+1) / (k+1-q)$. We refer to this condition by saying that the update times sequence $t = (t_i)_{i=1}^{\infty}$ should be G-*subgeometric*.

Next we introduce the notion of *cyclic* algorithms. Upper bounds on q_k presented in this paper, as well as in [1, 4], are all achieved by cyclic algorithms. Given a positive integer n and a real number $\gamma > 1$, a k-checkpoint algorithm $\text{ALG} = (t, p)$ is (n, γ)-*cyclic* if $t_{n+i} = \gamma \cdot t_i$ for all $i \geq 1$ and $p_i = p_{n+i}$ for all $i \geq k+1$. It has been observed in [4, Lemma 5] that any q-efficient (n, γ)-cyclic algorithm must satisfy $\gamma \leq G^n$ (to see this, apply subgeometry n times). An (n, γ)-cyclic algorithm is called (n, G)-*geometric* when $\gamma = G^n$ (and thus $t_{i+1} = G \cdot t_i$ for $i \geq k$).

We finish this subsection with two observations about the exponential growth of update times in efficient algorithms, relevant for upper and lower bounds on q_k.

The first one is an improvement of [4, Lemma 8]:

▶ **Fact 5.** *Any q-efficient k-checkpoint algorithm $\text{ALG} = (t, p)$ satisfies, without loss of generality, $t_{i+2} > t_i \cdot G$ for all $i \geq k$.*

An immediate corollary of Fact 5 is that $t_{i+j} > t_i \cdot G^{\lfloor j/2 \rfloor}$ for $j \geq 0$; in particular, $t_{i+j} > t_i \cdot G^2$ for $j \geq 4$. Our next observation says when we can get $t_{i+3} > t_i \cdot G^2$.

▶ **Fact 6.** *Let $S = (T_1, \ldots, T_k)$ be a snapshot of some q-efficient k-checkpoint algorithm $\text{ALG} = (t, p)$ such that $T_j = t_i$ and $T_{j+1} = t_{i+3}$ for some $j = 1, \ldots, k-1$. Thus, without loss of generality, $t_{i+3} > t_i \cdot G^2$.*

3   Optimal Algorithms for Small Values of k

3.1   Round-robin and $k \leq 5$

We now analyze the efficiency of the Round-Robin algorithm, which is geometric and always updates the oldest checkpoint (i.e., $p_i = 1$ for all $i \geq k+1$).[5] Besides serving as a first example, Round-Robin is optimal for $k \leq 3$ and will make an appearance within the asymptotically optimal algorithm Recursive of Section 4.

▶ **Proposition 7.** *The efficiency of k-checkpoint* Round-Robin *is $q = (k+1)\,r$, where r is the smallest real root of $x = (1-x)^{k-1}$.*

▶ **Remark.** Round-Robin is pretty bad for large k; indeed, $(k+1)\,r \approx \ln k - \ln \ln k$ is asymptotically inferior to the simple bound $q_k \leq 2$ from the introduction.

The case $k = 2$ is made obvious by Fact 3, since without loss of generality Round-Robin is the *only* 2-checkpoint algorithm to consider. Thus $q_2 = 1.5$.

▶ **Proposition 8.** *For $k = 3$ we have $q_3 = 4r_3 \approx 1.52786$, where $r_3 = \frac{3-\sqrt{5}}{2} \approx 0.38197$ is the smaller root of $x^2 - 3x + 1 = 0$.*

Proof. For the upper bound, Round-Robin is $4r_3$-efficient. For the lower bound, consider a $4r$-efficient 3-checkpoint algorithm and a snapshot $S_i = (x, y, t_i)$. By subgeometry we must have $t_i \leq y/(1-r) \leq x/(1-r)^2$ and for subinterval 1 to be compliant we need $x \leq rt_i$, which together imply $(1-r)^2 \leq r$, i.e., $r^2 - 3r + 1 \leq 0$. Thus $r \geq r_3$.   ◀

Round-Robin is no longer optimal for $k > 3$. Indeed, cyclic algorithms with better efficiency were described in [4, Figure 3] for $k = 4, 5, 6, 7, 8$. These provide upper bounds on $q_4, \ldots, q_8$, respectively. Nevertheless, no formal proof of optimality was provided.

▶ **Remark.** These algorithms were found by the use of linear programming, which is thoroughly discussed in the next subsection.

For $k = 4, 5$ the optimal algorithms are 2-cyclic; $k = 5$ is geometric while $k = 4$ is not.

▶ **Proposition 9.** *For $k = 4$ we have $q_4 = 5r_4 \approx 1.53989$, where $r_4 = (2 + 2\cos(2\pi/7))^{-1} \approx 0.307979$ is the smallest root of $x^3 - 5x^2 + 6x - 1 = 0$. Moreover, the efficiency of any geometric 4-checkpoint algorithm is at least $5\tilde{r}_4 \approx 1.58836$, where $\tilde{r}_4 \approx 0.31767 > r_4$ is the real root of $x^3 - 3x^2 + 4x - 1 = 0$.*

▶ **Proposition 10.** *For $k = 5$ we have $q_5 = 6r_5 \approx 1.47073$, where $r_5 \approx 0.24512$ is the (only) real root of $x^3 - 4x^2 + 5x - 1 = 0$.*

3.2   Casting the problem as a linear program

Fix $\lambda \geq 1$ and an update pattern $p = (p_i)_{i=k+1}^{\infty}$. Can we choose a sequence $t = (t_i)_{i=1}^{\infty}$ of update times such that the resulting k-checkpoint algorithm $\text{Alg} = (t, p)$ is λ-efficient?

Each snapshot S_i consists of a particular subset of the variables t, and using p we can determine exactly which. Furthermore, all constraints (e.g., monotonicity, subgeometry, compliance) can be expressed as linear inequalities. This gives rise to an infinite linear program $L = L(\lambda; p)$, which is feasible whenever a λ-efficient algorithm with the prescribed

---

[5] The case $k = 3$ of Round-Robin was considered in [4, Theorem 1] under the name Simple.

pattern p exists. Note that all constraints are homogeneous, so to avoid the zero solution we add the non-homogeneous condition $t_k = 1$.

In addition, we are not interested in solutions where t is bounded. This can happen, for instance, when $p_i = k - 1$ for all $i \geq k + 1$.[6] Luckily, by using Fact 5 we can restrict our attention to exponentially increasing sequences t, so we add to L the linear inequalities from Facts 5 and 6. Now L is feasible if and only if a λ-efficient algorithm with the prescribed pattern p exists; in other words, q_k is the infimum[7] over $\lambda \geq 1$ for which there exists a pattern p such that $L(\lambda; p)$ is feasible.

As an infinite program, L is not too convenient to work with. We can thus limit our attention to finite subprograms $L(\lambda; (p_{k+1}, \ldots, p_{k+n}))$ for some $n \in \mathbb{N}$, which only involve the $k + n$ variables $t_1, \ldots, t_{k+n}$ and the relevant $3k + 6n$ constraints. Finite subprograms can no longer ensure the existence of a λ-efficient algorithm, but can be used to prove lower bounds on q_k in the following way. Write $\Sigma = \{1, \ldots, k - 1\}$ and consider the set Σ^* of strings, i.e, finite sequences over Σ.

▶ **Definition.** A string $B \in \Sigma^*$ is called a λ-*witness* if $L(\lambda; B)$ is infeasible. A string set $\mathcal{B} \subset \Sigma^*$ is called *blocking* if any infinite sequence p over Σ contains some $B \in \mathcal{B}$ as a substring.

▶ **Fact 11.** *If there exists a blocking set of λ-witnesses for some $\lambda \geq 1$, then $q_k > \lambda$.*

▶ Remark. The lower bound of Fact 11 holds for all algorithms, cyclic or not.

We now describe a strategy to approximate q_k to arbitrary precision. For the lower bound we use Fact 11; for the upper bound, we limit our focus to cyclic algorithms. Given $\gamma > 1$ and a string $P \in \Sigma^*$ of length n, we can augment $L(\lambda; P)$ with k equality constraints $\{t_{i+n} = \gamma \cdot t_i\}_{i=1}^k$; call the resulting program $L^*(\lambda, \gamma; P)$. This is a finite linear program, which we can computationally solve given λ, γ, and P. Although $\gamma \leq G^n$ is not known to us, we can first compute an approximation $\tilde{\gamma}$ of γ by solving $L_{10n}(\gamma; P^{10})$, and then solve $L^*(\lambda, \tilde{\gamma}; P)$. Using binary search, we can compute a numerical approximation $\tilde{\lambda}$ of the minimal λ for which $L^*(\lambda, \tilde{\gamma}; P)$ is feasible. Lastly, we can enumerate short strings $P \in \Sigma^*$ in a BFS/DFS-esque manner and take the best $\tilde{\lambda}$ obtained.

To demonstrate this strategy, we computed $q_2, \ldots, q_{10}$ up to 7 decimal digits, using a Python program employing GLPK [6] via CVXOPT [2] (see Table 1; starred values of k are geometric algorithms).

At first it seems that Fact 11 cannot be used to pinpoint q_k exactly, since any finite blocking set $\mathcal{B}$ of $(q_k - \epsilon)$-witnesses for some $\epsilon > 0$ leaves an interval of uncertainty of length ϵ. The following proposition eliminates this uncertainty.

▶ **Proposition 12.** *For every string $B \in \Sigma^*$ there is finite set $\Lambda_B \subset \mathbb{R}$ such that the feasibility of $L(\lambda; B)$ for some $\lambda \geq 1$ only depends on the relative order between λ and members of Λ_B. In particular, there exists some $\epsilon > 0$ such that if $L(\lambda; B)$ is feasible and B is a $(\lambda - \epsilon)$-witness, then B is also a λ'-witness for all $\lambda - \epsilon < \lambda' < \lambda$.*

Proof. Fix $B \in \Sigma^*$. Treating λ as a parameter, note that the subprogram $L(\lambda; B)$ is feasible if and only if its feasible region, the convex polytope $\mathcal{P}(\lambda; B)$, is nonempty. Decreasing λ shrinks $\mathcal{P}(\lambda; B)$ until some critical λ_B for which $\mathcal{P}(\lambda_B; B)$ is reduced to a single vertex, at which a subset of the linear constraints are satisfied with equality. Hence λ_B is a solution of

---

[6] This may not seem a valid pattern to consider, given Fact 1; however, when solving a finite subprogram we might have to consider an arbitrarily long prefix of the pattern with no occurrences of 1.
[7] This infimum is actually a minimum, by [4, Theorem 8] and also by Proposition 12.

Table 1 Computationally-verified bounds on q_k for $2 \le k \le 10$.

k	q_k	γ	P	n	$|\mathcal{B}|$	$\max_{B \in \mathcal{B}} |B|$
2*	1.5	2	(1)	1	0	0
3*	1.5278641	1.618037	(1)	1	2	1
4	1.5398927	1.8019377	(1,3)	2	7	3
5*	1.4707341	1.7548777	(1,3)	2	36	13
6	1.5127400	3.627365	(1,2,3,1,3,5)	6	117	9
7	1.4974818	3.11201	(1,3,4,1,5,3)	6	559	10
8	1.4851548	10.712656	(1,2,4,7,5,3,1,7,5,3,7,1,4,2,4,5)	16	1698	14
9	1.4730721	3.2748095	(1,5,3,5,1,5,6,3)	8	5892	135
10	1.4678452	5.67943	(1,5,3,5,1,5,6,3,1,5,9,3,5,9)	14	32843	20

some polynomial equation determined by the relevant constraints. The set of constraints is finite, thus there are finitely many polynomial equations that can define λ_B, and we can take Λ_B as the set of all their roots. Now take ϵ to be smaller than the distance between any two distinct elements of Λ_B.   ◄

▶ Remark. Note that when ϵ is small enough, we can actually retrieve the polynomial equations defining λ and γ from the polytope $\mathcal{P}^* \left(\tilde{\lambda}, \tilde{\gamma}; B \right)$; using this method we get an algebraic representation of q_k rather than a rational approximation. To demonstrate, $q_9 = 10 r_9$, where $r_9 \approx 0.1473072131$ is the smallest real root of $x^8 - 7x^7 + 22x^6 - 40x^5 + 39x^4 - 17x^3 + 10x^2 - 8x + 1$.

4   Asymptotically Optimal Upper Bounds

In this section we describe a family of geometric k-checkpoint algorithms. Despite our experience from Table 1—that only for $k = 2, 3, 5$ optimal algorithms are geometric—this family is rich enough to be asymptotically optimal, i.e., $(1 + o(1)) q_k$-efficient.

4.1   A recursive geometric algorithm

Fix a real number $G > 1$ and an integer $m \ge 0$. We describe a k-checkpoint algorithm RECURSIVE(G, K), where K is an $(m + 2)$-subset $\{0, \dots, k\}$ whose elements are

$$k = k_0 > k_1 > k_2 > \cdots > k_m > k_{m+1} = 0.$$

RECURSIVE(G, K) is $(2^m, G)$-geometric, and its update pattern p is defined as $p_{k+i} = 1 + k_{\mu(i)+1}$, where $\mu(i)$ is the largest $\mu \le m$ for which 2^μ divides i. It is easy to see that p is 2^m-periodic, and we can just refer to $P = (p_{k+i})_{i=1}^{2^m}$. As per Remark 2, via rebasing there is no need to define the initial snapshot S_k explicitly.

▶ **Example.** For $K = \{0, 2, 4, 9, 19\}$ we get $P = (10, 5, 10, 3, 10, 5, 10, 1)$.

True to its name, RECURSIVE(G, K) can be viewed also as a recursive algorithm: the base case $m = 0$ (i.e., $K = \{0, k\}$) is simply k-checkpoint ROUND-ROBIN; for $m \ge 1$, RECURSIVE(G, K) alternates between updating the $(k_1 + 1)$-st oldest checkpoint and between acting according to the inner k_1-checkpoint algorithm RECURSIVE$\left(G^2, K \setminus \{k\}\right)$.

Let us elaborate a bit more on the recursive step. In every snapshot $S_i = (T_1, \dots, T_k)$ we have $T_j = G^{i+j}$ for $k_1 + 1 \le j \le k$ since we never update checkpoints younger than $k_1 + 1$. In every *odd* snapshot S_i we have just updated the $(k_1 + 1)$-st oldest checkpoint,

so $T_{k_1} = G^{i+k_1-1}$ while $T_{k_1+1} = G^{i+k_1+1}$. This means that $\log_G T_j$ for $j = 1, \ldots, k_1$ all have the same parity as $i + k_1 - 1$ in any snapshot S_i. We thus treat $S' = (T_1, \ldots, T_{k_1})$ as a snapshot of a k_1-checkpoint algorithm, which operates at half speed and never sees half of the checkpoints. The inner algorithm can rightfully be called $\text{RECURSIVE}\left(G^2, K \setminus \{k\}\right)$, as the common ratio of the update times sequence for the checkpoints that do make it to the inner algorithm is G^2, and taking only the even locations of P yields a 2^{m-1} periodic sequence p' such that $p'_{k+i} = p_{k+2i} = 1 + k_{\mu(2i)+1} = 1 + k_{\mu(i)+2}$.

4.2 Analyzing the recursive algorithm

First we determine exactly how efficient $\text{RECURSIVE}(G, K)$ can be for any G and K, and then we work with a particular choice.

Denote by $r(G, K)$ the maximum of

$$1 - G^{-1}; \tag{1a}$$

$$\max \left\{ G^{-e(\ell)} \left(G^{2^\ell} - G^{-2^\ell} \right) \right\}_{\ell=0}^{m-1}; \text{ and} \tag{1b}$$

$$G^{2^m - e(m)}, \tag{1c}$$

where $e(\ell) = \sum_{j=0}^{\ell} 2^j (k_j - k_{j+1})$ for $\ell = 0, 1, \ldots, m$.

▶ **Theorem 13.** *Given G and K, the efficiency of $\text{RECURSIVE}(G, K)$ is $(k+1)r(G, K)$.*

Given an integer $k \geq 2$, let $m = \lfloor \log_2 k \rfloor - 1$. Define $K^* = \{k_0, \ldots, k_{m+1}\}$ by $k_j = \lfloor 2^{-j} k \rfloor$ for $j = 0, \ldots, m$ and $k_{m+1} = 0$. Note that $k_0 = k$ and that $k_m \in \{2, 3\}$.

▶ **Theorem 14.** *$\text{RECURSIVE}(G, K^*)$ is q-efficient for large enough k, where $G = e^{q/(k+1)}$ and $q = \left(1 + \dfrac{3}{\log_2 k}\right) \dfrac{k+1}{k} \ln 4 = (1 + o(1)) \ln 4$.*

Proof. By Theorem 13, it suffices to verify that $(k+1)r(G, K^*) < q$, that is, $r(G, K^*) \leq \ln G$ for sufficiently large k. Clearly (1a) holds since $1 - G^{-1} < \ln G$ for all $G > 1$. It remains to verify (1b) and (1c), handled by Propositions 15 and 16 respectively. ◀

▶ **Proposition 15.** *For $k \geq 2^{13}$ and G as above, $G^{-e(\ell)} \left(G^{2^\ell} - G^{-2^\ell} \right) < \ln G$ for all $\ell = 0, \ldots, m - 1$.*

▶ **Proposition 16.** *For $k \geq 5$ and G as above, $G^{2^m - e(m)} < \ln G$.*

▶ Remark. Theorem 14 chooses G suboptimally. Empirical evidence shows that, for all $k \geq 2$, the optimal $G = G^*$ for $\text{RECURSIVE}(G, K^*)$ satisfies (1a) and one of (1b) and (1c). In other words, it is the smallest root of either $1 - x^{-1} = x^{2^m - e(m)}$ or $1 - x^{-1} = x^{-e(\ell)} \left(x^{2^\ell} - x^{-2^\ell} \right)$ for some $\ell = 0, \ldots, m - 1$.

▶ Remark. With additional effort the constant 3 in Theorem 14 can be improved by a factor of almost 6 to $\tau := -\log_2 \ln 2 \approx 0.53$. The major obstacle is that cases $\ell = m - 2$ and $\ell = m - 1$ of (1b) need to be done separately since the appropriate $f(x, z)$ in the proof of Proposition 15 is negative for $z < \log_2 (\ln 4 / (1 - \ln 2)) \approx 2.1756$. No proof is possible for $\tau' < \tau$ since then (1c) would be violated for large enough $k = 2^{m+2} - 1$.

▶ Remark. We verified that the algorithm $\text{RECURSIVE}(G^*, K^*)$ is $(1 + \tau / \log_2 k) \frac{k+1}{k} \ln 4$-efficient for $2 \leq k \leq 2^{13}$ as well.

5   Asymptotically Optimal Lower Bounds

In this section we prove lower bounds on q_k, focusing on asymptotic lower bounds in which k grows to infinity.

We start by reproving the simple asymptotic lower bound $q_k \geq 2 - \ln 2 - o(1)$ of [4, Theorem 6], and then improve it to $q_k \geq \ln 4$, which is asymptotically optimal via the matching upper bound of Section 4.

5.1   Stability and bounding expressions

Obtaining lower bounds requires viewing the problem from a different perspective. It will sometimes be more convenient to refer to a certain physical checkpoint, without considering its temporary freshness index p in the checkpoint sequence at some snapshot S (which is variable and depends on S).

Given a k-checkpoint algorithm, we define a function $BE(s)$ and use it to bound its efficiency from below. The parameter s is related to the notion of stability, which we now define.

▶ **Definition.** Fix a k-checkpoint algorithm. A checkpoint updated at time T is called s-stable, for some $s = 1, \ldots, k-1$, if at least s previous checkpoints are updated before the next time it is updated.

By Fact 1 we can assume all checkpoints get updated eventually; this means that in a snapshot $S = (T_1, \ldots, T_k)$, where T_k is a time by which all checkpoints have been updated from the initial snapshot, we have that the checkpoint updated at time T_{k-s} is s-stable for $s = 1, \ldots, k-1$.

For convenience, the proofs in this section assume the update times sequence is normalized by a constant. This is captured by the following definition.

▶ **Definition.** A k-checkpoint algorithm is called s-normalized if an s-stable checkpoint is updated at time $R_0 = 1$.

Given an s-normalized k-checkpoint algorithm, we define a sequence of times $1 = R_0 < R_1 < \cdots < R_s$ as follows: R_i for $i \geq 1$ is the time at which the i-th checkpoint is *removed* from $(0, 1]$. In other words, R_1 is the time $T > R_0$ at which some checkpoint is updated; R_2 is the time $T > R_1$ at which we update the next checkpoint that was previously updated in $(0, 1]$ (but not at $T > 1$), and so forth. Note that the checkpoint updated at time R_0 is not updated at any time R_i for $i = 1, \ldots, s$ by the definition of stability. Now we are ready to define $BE(s)$.

▶ **Definition.** The T-truncated bounding expression of an s-normalized k-checkpoint algorithm is $BE_T(s) = \sum_{i=1}^{s} U_i$, where $U_i = \min\{T, R_i\}$.

The bounding expression plays a crucial role in proving lower bounds, based on Proposition 17 below. We note that the truncated bounding expression only depends on the algorithm's behavior until time T, and hence the bounds that can be obtained from it are not tight for $k > 3$. Nevertheless, the lower bound we obtain using BE_2 in Corollary 22 is asymptotically optimal, since the gap between it and the upper bound of Theorem 14 tends to zero as k grows to infinity.

▶ **Remark.** It is possible to analyze BE_T beyond $T = 2$ and obtain tight lower bounds for larger values of k. However, there is no asymptotic improvement and the analysis becomes increasingly more technical as k grows.

5.2   Asymptotic lower bound of $2 - \ln 2 \approx 1.3068$

To simplify the analysis, we assume k is even. It can be extended to cover odd values of k as well, but this gives no asymptotic improvement since $q_{k+1} \leq q_k \cdot \frac{k+2}{k+1}$ for all k, so we only lose an error term of $O\left(1/k\right)$, which is of the same order as the error terms in Corollaries 20 and 22.

To simplify our notation we write $b = q/(k+1)$ throughout this section.

▶ **Proposition 17.** *Any $(k/2)$-normalized $b(k+1)$-efficient k-checkpoint algorithm satisfies $BE_2(k/2) \geq 1/b$.*

Proof. At time $R_0 = 1$, the time interval $(0, 1]$ contains k subintervals of length $\leq b$, giving rise to the inequality $b \cdot k \geq 1$. At time R_1, a checkpoint is removed from $(0, 1]$ and it now contains one subinterval of length $\leq b \cdot R_1$ (two previous subintervals, each of length $\leq b$, were merged), and $k - 2$ subintervals of length $\leq b$, giving $b \cdot (k - 2 + R_1) \geq 1$.

At time R_2, an additional checkpoint is removed from the time interval $(0, 1]$, hence it must contain an subinterval of length $\leq b \cdot R_2$ formed by merging two previous subintervals. We obtain $b \cdot (k - 4 + R_1 + R_2) \geq 1$, since the remaining $k - 3$ subintervals must include $k - 4$ subintervals of length $\leq b$ and one (additional) subinterval of length at most $\leq b \cdot R_1$. Note that this claim holds regardless of which checkpoint is updated at R_2, and it holds in particular in case one of the subintervals merged at time R_2 contains the subintervals merged at R_1 (in fact, this case gives the stronger inequality $b \cdot (k - 3 + R_2) \geq 1$).

In general, for $j = 1, 2, \ldots, k/2$, at time R_j the time interval $(0, 1]$ must contain j distinct subintervals of lengths $\leq b \cdot R_i$ for $i = 1, 2, \ldots, j$, and $k - 2j$ subintervals of length $\leq b$. This gives the inequality $k - 2j + \sum_{i=1}^{j} R_i \geq 1/b$.

Let $j \leq k/2$ be the largest index such that $R_j \leq 2$, so $U_i = R_i$ for all $1 \leq i \leq j$ and $U_i = 2$ for all $j < i \leq k/2$. Now at time R_j we have

$$1/b \leq k - 2j + \sum_{i=1}^{j} R_i = (k/2 - j) \cdot 2 + \sum_{i=1}^{j} U_i = \sum_{i=j+1}^{k/2} U_i + \sum_{i=1}^{j} U_i = BE_2(k/2). \qquad ◀$$

Now we need an upper bound on the bounding expression. For the simpler lower bound of $2 - \ln 2$ we use the following proposition.

▶ **Proposition 18.** *Any $(k/2)$-normalized $b(k+1)$-efficient k-checkpoint algorithm satisfies $R_i \leq 1/\left(1 - bi\right)$ for $i = 1, \ldots, k/2$.*

Proof. At time $T = 1/\left(1 - bi\right)$, all subintervals are of length at most $bT = b/\left(1 - bi\right)$. Since $T - R_0 = 1/\left(1 - bi\right) - 1 = bi/\left(1 - bi\right)$, for any $\epsilon > 0$ the time interval $(R_0, T + \epsilon]$ must consist of at least $i + 1$ subintervals, implying that the i-th checkpoint was removed from the time interval $(0, 1]$ by time T. ◀

▶ **Proposition 19.** *Let $b < \frac{1}{2}$. Any $(k/2)$-normalized $b(k+1)$-efficient k-checkpoint algorithm satisfies $b \cdot BE_2(k/2) \leq \ln 2 + b/(1 - 2b) + bk - 1$.*

▶ **Corollary 20.** *For all even $k \geq 4$ we have $q_k \geq 2 - \ln 2 - o\left(1\right)$.*

Proof. Fix a $(k/2)$-normalized q_k-efficient k-checkpoint algorithm, and let $b = q_k/(k+1) < \frac{1}{2}$. By Propositions 17 and 19 we have

$$q_k = bk + b \geq 2 - \ln 2 - \frac{b}{1 - 2b} + b = 2 - \ln 2 - \frac{2b^2}{1 - 2b} \geq 2 - \ln 2 - \frac{8}{(k+1)(k-3)}. \qquad ◀$$

5.3   Improved asymptotic lower bound of $\ln 4 \approx 1.3863$

We now improve the asymptotic lower bound to $\ln 4$. This result is a simple corollary of the following lemma, which gives a tighter upper bound on the bounding expression. Recall that $q = b(k+1)$ and thus $G = (k+1)/(k+1-q) = 1/(1-b)$.

▶ **Lemma 21.** *For any s-normalized $b(k+1)$-efficient k-checkpoint algorithm such that $1 \leq s \leq k/2$ and $G^{k/2} \leq 2$ we have $b \cdot BE_2(s) \leq G^s - 1$.*

▶ **Corollary 22.** *For all even $k \geq 2$ we have $(1 - q_k/(k+1))^{-k/2} \geq 2$. In particular, $q_k > \ln 4$.*

Proof. Write $b = q_k/(k+1)$ and assume for the sake of contradiction that $G^{k/2} < 2$. By Lemma 21 and Proposition 17 we have $1 \leq b \cdot BE_2(k/2) \leq G^{k/2} - 1$ for a $k/2$-normalized q_k-efficient k-checkpoint algorithm, so $G^{k/2} \geq 2$, contradicting our assumption. Now

$$\frac{q_k}{k+1} = b \geq 1 - 2^{-2/k} = 1 - e^{-(\ln 4)/k} \geq \frac{\ln 4}{k} - \frac{1}{2}\left(\frac{\ln 4}{k}\right)^2 = \left(1 - \frac{\ln 2}{k}\right)\frac{\ln 4}{k},$$

hence $q_k \geq \left(1 + (1 - \ln 2)/k - (\ln 2)/k^2\right)\ln 4 > \ln 4$. The last inequality is true when $k > \ln 2/(1 - \ln 2) \approx 2.26$, but we already know that $q_2 = 1.5 > \ln 4$.   ◀

6   Concluding Remarks and Open Problems

In this paper we solved the main open problem in online checkpointing algorithms, which is to find tight asymptotic upper and lower bounds on their achievable efficiency. In addition, we developed efficient techniques for determining tight upper and lower bounds on q_k for small values of k, which enabled us to develop provably optimal concrete algorithms for all $k \leq 10$. However, determining the values of q_k for larger values of k remains a computationally challenging problem, and finding more efficient ways to compute these values remains an interesting open problem.

References

1   Lauri Ahlroth, Olli Pottonen, and André Schumacher. Approximately Uniform Online Checkpointing with Bounded Memory. *Algorithmica*, 67(2):234–246, 2013. `doi:10.1007/s00453-013-9772-5`.

2   M.S. Andersen, J. Dahl, and L. Vandenberghe. CVXOPT: A Python package for convex optimization, 2016. version 1.1.9. Available at `http://cvxopt.org`.

3   Marshall W. Bern, Daniel H. Greene, Arvind Raghunathan, and Madhu Sudan. On-Line Algorithms for Locating Checkpoints. *Algorithmica*, 11(1):33–52, 1994. `doi:10.1007/BF01294262`.

4   Karl Bringmann, Benjamin Doerr, Adrian Neumann, and Jakub Sliacan. Online Checkpointing with Improved Worst-Case Guarantees. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 255–266, 2013. `doi:10.1007/978-3-642-39206-1_22`.

5   K. Mani Chandy and Chittoor V. Ramamoorthy. Rollback and Recovery Strategies for Computer Programs. *IEEE Trans. Computers*, 21(6):546–556, 1972. `doi:10.1109/TC.1972.5009007`.

6   Free Software Foundation. Gnu linear programming kit, 2012. Version 4.61, `http://www.gnu.org/software/glpk/`.

7   Erol Gelenbe. On the Optimum Checkpoint Interval. *J. ACM*, 26(2):259–270, 1979. `doi:10.1145/322123.322131`.

8   Sam Toueg and Özalp Babaoglu. On the Optimum Checkpoint Selection Problem. *SIAM J. Comput.*, 13(3):630–649, 1984. `doi:10.1137/0213039`.

Table 2 Computationally-verified upper bounds on q_k for $11 \leq k \leq 20$.

k	λ	γ	P	n
11	1.4650841	8.190656	(1,3,5,6,1,6,2,10,6,3,6,1,6,2,6,3,9,6)	18
12	1.4668421	8.862576	(1,2,3,5,6,7,1,2,6,3,6,7,1,2,6,3,6,9,7)	19
13	1.4592320	2.94	(1,3,6,7,4,7,1,7,8,3)	10
14	1.4570046	58.6	(1,4,2,6,7,4,7,8,1,8,2,3,7,12,4,7,8,1,4,7,2,7,8,4,13,8, 1,8,4,2,7,4,7,8,1,8,4,2,7,12,4,7,13,8)	44
15	1.4487459	2.104027	(1,2,7,8,4,8,9,5)	8
16	1.4487597	8.46	(1,2,4,7,8,9,5,9,1,2,8,4,8,9,5,9,1,2,8,4,8,13,9,5,9)	25
17	1.4593611	1.694884	(1,9,5,3,14,8,9)	7
18	1.4575670	2.57	(1,8,9,5,9,10,2,5,9,10,3,5)	12
19	1.4592194	2.45	(1,9,5,9,10,2,5,9,10,11,3,5)	12
20	1.4696048	13.3	(1,5,9,10,2,5,9,10,11,3,5,10,1,5,9,10,11,2,5,10,11,3,5,10, 1,5,9,10,6,2,9,10,11,3,5,10)	36

A Tables

The best algorithms our LP approach of Section 3.2 found for $k = 11, 12, \ldots, 20$ are described in Table 2. These are (perhaps non-tight) upper bounds on $q_{11}, \ldots, q_{20}$. Observe how some of the patterns are reminiscent of the pattern used in the algorithm RECURSIVE of Section 4.