Chapter 1

Dynamic $\Omega(\lg n)$ Bounds

In this chapter, we prove our first lower bounds: we show that the partial sums and dynamic connectivity problems require $\Omega(\lg n)$ time per operation. We introduce the proof technique using the partial sums problem; dynamic connectivity requires two additional tricks, described in §1.6 and §1.7. The proofs are surprisingly simple and clean, in contrast to the fact that proving any $\Omega(\lg n)$ bound was a well-known open problem for 15 years before our paper [PD06].

1.1 Partial Sums: The Hard Instance

It will pay to consider the partial sums problem in a more abstract setting, namely over an arbitrary group $\mathbb{G}$. Remember that the problem asked to maintain an array $A[1 \,..\, n]$, initialized to zeroes (the group identity), under the following operations:
UPDATE(k, Δ)**:** modify $A[k] \leftarrow \Delta$, where $\Delta \in \mathbb{G}$.
SUM(k)**:** returns the partial sum $\sum_{i=1}^{k} A[i]$.

Our proof works for any choice of $\mathbb{G}$. In the cell-probe model with w-bit cells, the most natural choice of $\mathbb{G}$ is $\mathbb{Z}/2^w\mathbb{Z}$, i.e. integer arithmetic modulo 2^w. In this case, the argument Δ of an update is a machine word. Letting $\delta = \lg|\mathbb{G}|$, our proof will show that any data structure requires an average running time of $\Omega(\frac{\delta}{w} \cdot n \lg n)$ to execute a sequence of n updates and n queries chosen from a particular distribution. If $\delta = w$, we obtain an amortized $\Omega(\lg n)$ bound per operation.

The hard instance is described by a permutation π of size n, and a sequence $\langle \Delta_1, \ldots, \Delta_n \rangle \in \mathbb{G}^n$. Each Δ_i is chosen independently and uniformly at random from $\mathbb{G}$; we defer the choice of π until later. For t from 1 to n, the hard instance issues two operations: the query SUM$(\pi(t))$, followed by UPDATE$(\pi(t), \Delta_t)$. We call t the "time," saying, for instance, that SUM$(\pi(t))$ occurs at time t.

A very useful visualization of an instance is as a two-dimensional chart, with time on one axis, and the index in A on the other axis. The answer to a query SUM$(\pi(t))$ is the sum of the update points in the rectangle $[0, t] \times [0, \pi(t)]$; these are the updates which have already occurred, and affect indices relevant to the partial sum. See Figure 1-1 (a).

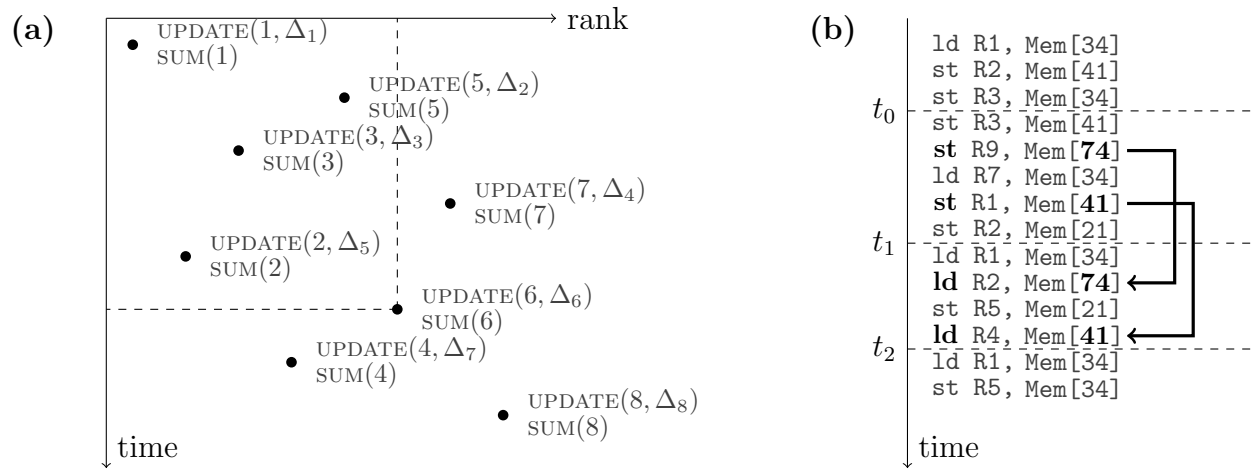

Figure 1-1: (a) An instance of the partial sums problem. The query SUM(6) occurring at time 6 has the answer $\Delta_1+\Delta_2+\Delta_3+\Delta_5$. (b) The execution of a hypothetical cell-probe algorithm. $IT(t_0, t_1, t_2)$ consists of cells 41 and 74.

1.2 Information Transfer

Let $t_0 < t_1 < t_2$, where t_0 and t_2 are valid time values, and t_1 is non-integral to avoid ties. These time stamps define two adjacent intervals of operations: the time intervals $[t_0, t_1]$ and $[t_1, t_2]$; we will be preoccupied by the interaction between these time intervals. Since the algorithm cannot maintain state between operations, such interaction can only be caused by the algorithm writing a cell during the first interval and reading it during the second.

Definition 1.1. *The* information transfer $IT(t_0, t_1, t_2)$ *is the set of memory locations which:*

- *were read at a time* $t_r \in [t_1, t_2]$.
- *were written at a time* $t_w \in [t_0, t_1]$, *and not overwritten during* $[t_w + 1, t_r]$.

The definition is illustrated in Figure 1-1 (b). Observe that the information transfer is a function of the algorithm, the permutation π, and the sequence Δ.

For now, let us concentrate on bounding $|IT(t_0, t_1, t_2)|$, ignoring the question of how this might be useful. Intuitively, any dependence of the queries from $[t_1, t_2]$ on updates from the interval $[t_0, t_1]$ must come from the information in the cells $IT(t_0, t_1, t_2)$. Indeed, $IT(t_0, t_1, t_2)$ captures the only possible information flow between the intervals: an update happening during $[t_0, t_1]$ cannot be reflected in a cell written before time t_0.

Let us formalize this intuition. We break the random sequence $\langle \Delta_1, \ldots, \Delta_n \rangle$ into the sequence $\Delta_{[t_0,t_1]} = \langle \Delta_{t_0}, \ldots, \Delta_{\lfloor t_1 \rfloor} \rangle$, and $\Delta^\star$ containing all other values. The values in $\Delta^\star$ are uninteresting to our analysis, so fix them to some arbitrary $\mathbf{\Delta}^\star$. Let A_t be the answer of the query SUM$(\pi(t))$ at time t. We write $A_{[t_1,t_2]} = \langle A_{\lceil t_1 \rceil}, \ldots, A_{t_2} \rangle$ for the answers to the queries in the second interval.

In information theoretic terms, the observation that all dependence of the interval $[t_1, t_2]$ on the interval $[t_0, t_1]$ is captured by the information transfer, can be reformulated as saying

that the *entropy* of the observable outputs of interval $[t_1, t_2]$ (i.e., the query results $A_{[t_1,t_2]}$) is bounded by the information transfer:

Lemma 1.2. $H\big(A_{[t_1,t_2]} \;\big|\; \Delta^\star = \mathbf{\Delta}^\star\big) \;\leq\; w \;+\; 2w \cdot \mathbf{E}\big[|IT(t_0,t_1,t_2)| \;\big|\; \Delta^\star = \mathbf{\Delta}^\star\big]$.

Proof. The bound follows by proposing an *encoding* for $A_{[t_1,t_2]}$, since the entropy is upper bounded by the average length of any encoding. Our encoding is essentially the information transfer; formally, it stores:

- first, the cardinality $|IT(t_0,t_1,t_2)|$, in order to make the encoding prefix free.
- the *address* of each cell; an address is at most w bits in our model.
- the *contents* of each cell at time t_1, which takes w bits per cell.

The average length of the encoding is $w \;+\; 2w \cdot \mathbf{E}\big[|IT(t_0,t_1,t_2)| \;\big|\; \Delta^\star = \mathbf{\Delta}^\star\big]$ bits, as needed. To finish the proof, we must show that the information transfer actually encodes $A_{[t_1,t_2]}$; that is, we must give a *decoding algorithm* that recovers $A_{[t_1,t_2]}$ from $IT(t_0,t_1,t_2)$.

Our decoding algorithm begins by simulating the data structure during the time period $[1, t_0 - 1]$; this is possible because $\Delta^\star$ is fixed, so all operations before time t_0 are known. It then *skips* the time period $[t_0, t_1]$, and simulates the data structure again during the time period $[t_1, t_2]$. Of course, simulating the time period $[t_1, t_2]$ recovers the answers $A_{[t_1,t_2]}$, which is what we wanted to do.

To see why it is possible to simulate $[t_1, t_2]$, consider a read instruction executed by a data structure operation during $[t_1, t_2]$. Depending on the time t_w when the cell was last written, we have the following cases:

$t_w > t_1$**:** We can recognize this case by maintaining a list of memory locations written during the simulation; the data is immediately available.

$t_0 < t_w < t_1$**:** We can recognize this case by examining the set of addresses in the encoding; the cell contents can be read from the encoding.

$t_w < t_0$**:** This is the default case, if the cell doesn't satisfy the previous conditions. The contents of the cell is determined from the state of the memory upon finishing the first simulation up to time $t_0 - 1$. □

1.3 Interleaves

In the previous section, we showed an upper bound on the dependence of $[t_1, t_2]$ on $[t_0, t_1]$; we now aim to give a lower bound. Refer to the example in Figure 1-2 (a). The information that the queries in $[t_1, t_2]$ *need to know* about the updates in $[t_0, t_1]$ is the sequence $\big\langle \Delta_6,\ \Delta_6 + \Delta_3 + \Delta_4,\ \Delta_6 + \Delta_3 + \Delta_4 + \Delta_5 \big\rangle$. Equivalently, the queries need to know $\big\langle \Delta_6,\ \Delta_3 + \Delta_4,\ \Delta_5 \big\rangle$, which are three independent random variables, uniformly distributed in the group $\mathbb{G}$.

This required information comes from *interleaves* between the update indices in $[t_0, t_1]$, on the one hand, and the query indices in $[t_1, t_2]$, on the other. See Figure 1-2 (b).

Definition 1.3. *If one sorts the set* $\{\pi(t_0), \ldots, \pi(t_2)\}$*, the interleave number* $IL(t_0,t_1,t_2)$ *is defined as the number of transitions between a value* $\pi(i)$ *with* $i < t_1$*, and a consecutive value* $\pi(j)$ *with* $j > t_1$*.*

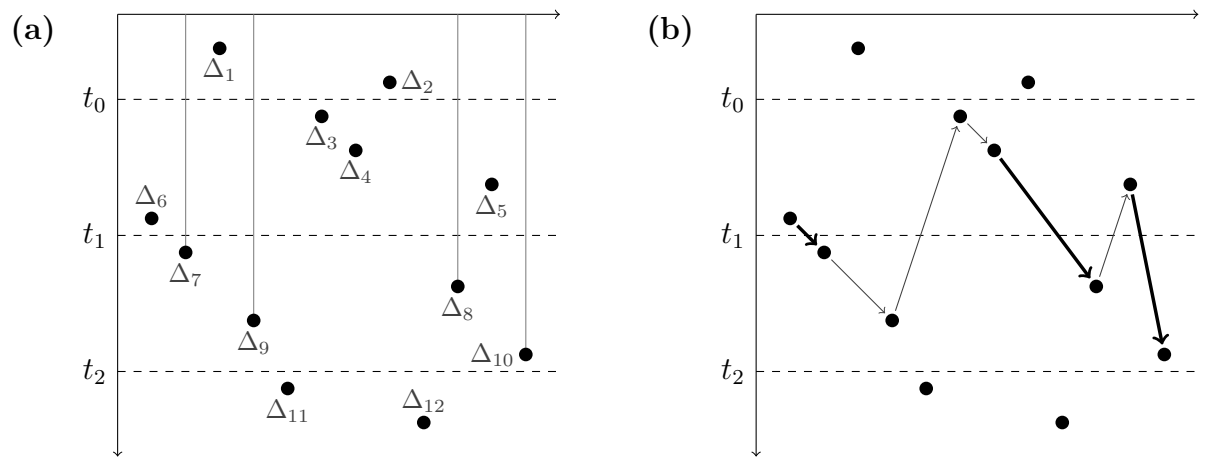

Figure 1-2: (a) The vertical lines describe the information that the queries in $[t_1, t_2]$ from the updates in $[t_0, t_1]$. (b) The interleave number $IL(t_0, t_1, t_2)$ is the number of down arrows crossing t_1, where arrows indicate left-to-right order.

The interleave number is only a function of π. Figure 1-2 suggests that interleaves between two intervals cause a large dependence of the queries $A_{[t_1,t_2]}$ on the updates $\Delta_{[t_1,t_2]}$, i.e. $A_{[t_1,t_2]}$ has large conditional entropy, even if all updates outside $\Delta_{[t_1,t_2]}$ are fixed:

Lemma 1.4. $H\big(A_{[t_1,t_2]} \;\big|\; \Delta^\star = \mathbf{\Delta}^\star\big) \;=\; \delta \cdot IL(t_0, t_1, t_2)$.

Proof. Each answer in $A_{[t_1,t_2]}$ is a sum of some random variables from $\Delta_{[t_0,t_1]}$, plus a constant that depends on the fixed $\Delta^\star$. Consider the indices $L = \{\pi(t_0), \ldots, \pi(\lfloor t_1 \rfloor)\}$ from the first interval, and $R = \{\pi(\lceil t_1 \rceil), \ldots, \pi(t_2)\}$ from the second interval. Relabel the indices of R as $r_1 < r_2 < \cdots$ and consider these r_i's in order:

- If $L \cap [r_{i-1}, r_i] = \emptyset$, the answer to $\textsc{sum}(r_i)$ is the same as for $\textsc{sum}(r_{i-1})$, except for a different constant term. The answer to $\textsc{sum}(r_i)$ contributes nothing to the entropy.
- Otherwise, the answer to $\textsc{sum}(r_i)$ is a random variable independent of all previous answers, due to the addition of random Δ's to indices $L \cap [r_{i-1}, r_i]$. This random variable is uniformly distributed in $\mathbb{G}$, so it contributes δ bits of entropy. □

Comparing Lemmas 1.4 and 1.2, we see that $\mathbf{E}\big[|IT(t_0, t_1, t_2)| \;\big|\; \Delta^\star = \mathbf{\Delta}^\star\big] \geq \frac{\delta}{2w} \cdot IL(t_0, t_1, t_2) - 1$ for any fixed $\mathbf{\Delta}^\star$. By taking expectation over $\Delta^\star$, we have:

Corollary 1.5. *For any fixed π, $t_0 < t_1 < t_2$, and any algorithm solving the partial sums problem, we have $\mathbf{E}_\Delta\big[|IT(t_0, t_1, t_2)|\big] \;\geq\; \frac{\delta}{2w} \cdot IL(t_0, t_1, t_2) - 1$.*

1.4 A Tree For The Lower Bound

The final step of the algorithm is to consider the information transfer between many pairs of intervals, and piece together the lower bounds from Corollary 1.5 into one lower bound

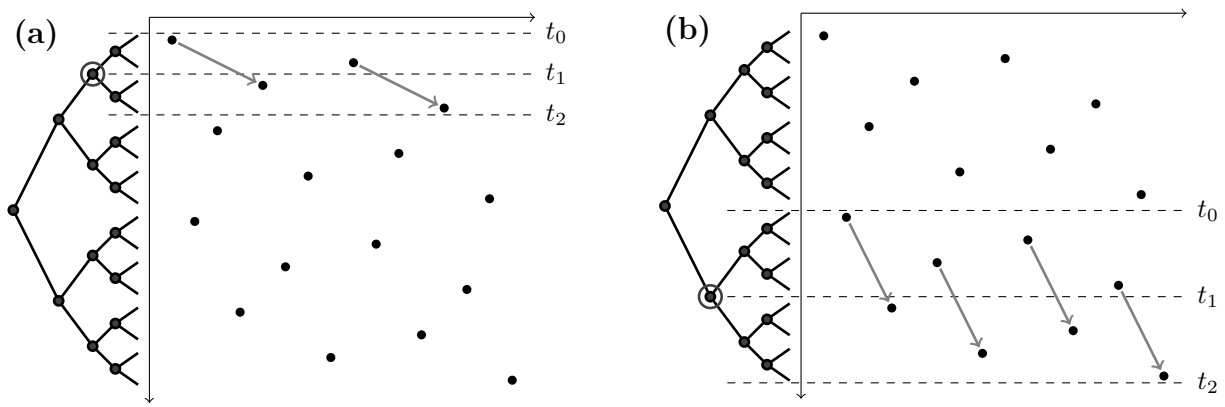

Figure 1-3: The bit-reversal permutation of size $n = 16$, and the lower-bound tree over the time axis. Each node has a maximal possible interleave: e.g. 2 in (a), and 4 in (b).

for the total running time of the data structure. The main trick for putting together these lower bounds is to consider a lower-bound tree $\mathcal{T}$: an arbitrary binary tree with n leaves, where each leaf denotes a time unit (a query and update pair). In other words, $\mathcal{T}$ is built "over the time axis," as in Figure 1-3.

For each internal node v of $\mathcal{T}$, we consider the time interval $[t_0, t_1]$ spanning the left subtree, and the interval $[t_1, t_2]$ spanning the right subtree. We then define:

- the information transfer *through* the node: $IT(v) = |IT(t_0, t_1, t_2)|$. Essentially, $IT(v)$ counts the cells written in the left subtree of v, and read in the right subtree.
- the interleave at the node: $IL(v) = IL(t_0, t_1, t_2)$.

Theorem 1.6. *For any algorithm and fixed π, the expected running time of the algorithm over a random sequence Δ is at least $\frac{\delta}{2w} \sum_{v \in \mathcal{T}} IL(v) \; - \; n$.*

Proof. First, observe that on any problem instance (any fixed Δ), the number of read instructions executed by the algorithm is at least $\sum_{v \in \mathcal{T}} IT(v)$. Indeed, for each read instruction, let t_r be the time it is executed, and $t_w \le t_r$ be the time when the cell was last written. If $t_r = t_w$, we can ignore this trivial read. Otherwise, this read instruction appears in the information transfer through exactly one node: the lowest common ancestor of t_w and t_r. Thus, $\sum_v IT(v)$ never double-counts a read instruction.

Now we apply Corollary 1.5 to each node, concluding that for each v, $\mathbf{E}_\Delta[IT(v)] \ge \frac{\delta}{2w} \cdot IL(v) - 1$. Thus, the total expected running time is at least $\frac{\delta}{2w} \sum_v IL(v) - (n-1)$. It is important to note that each lower bound for $|IT(v)|$ applies to the expected value under the same distribution (a uniformly random sequence Δ). Thus we may sum up these lower bounds to get a lower bound on the entire running time, using linearity of expectation. □

To complete our lower bound, it remains to design an access sequence π that has high total interleave, $\sum_{v \in \mathcal{T}} IL(v) = \Omega(n \lg n)$, for some lower-bound tree $\mathcal{T}$. From now on, assume n is a power of 2, and let $\mathcal{T}$ be a perfect binary tree.

Claim 1.7. *If π is a uniformly random permutation,* $\mathbf{E}_\pi\big[\sum_{v\in\mathcal{T}} IL(v)\big] = \Omega(n \lg n)$.

Proof. Consider a node v with $2k$ leaves in its subtree, and let S be the set of indices touched in v's subtree, i.e. $S = \{\pi(t_0), \ldots, \pi(t_2)\}$. The interleave at v is the number of down arrows crossing from the left subtree to the right subtree, when S is sorted; see Figure 1-2 (b) and Figure 1-3. For two indices $j_1 < j_2$ that are consecutive in S, the probability that j_1 is touched in the left subtree, and j_2 is touched in the right subtree will be $\frac{k}{2k} \cdot \frac{k}{2k-1} > \frac{1}{4}$. By linearity of expectation over the $2k-1$ arrows, $\mathbf{E}_\pi[IL(v)] = (2k-1)\cdot\frac{k}{2k}\cdot\frac{k}{2k-1} = \frac{k}{2}$. Summing up over all internal nodes v gives $\mathbf{E}_\pi\big[\sum_v IL(v)\big] = \frac{1}{4} n \log_2 n$. □

Thus, any algorithm requires $\Omega(\frac{\delta}{w}\cdot n \lg n)$ cell probes in expectation on problem instances given by random Δ and random π. This shows our $\Omega(\lg n)$ amortized lower bound for $\delta = w$.

1.5 The Bit-Reversal Permutation

An interesting alternative to choosing π randomly, is to design a worst-case π that *maximizes* the total interleave $\sum_{v\in\mathcal{T}} IL(v)$. We construct π recursively. Assume π' is the worst-case permutation of size n. Then, we *shuffle* two copies of π' to get a permutation π of size $2n$. Formally:

$$\pi = \big\langle 2\pi'(1) - 1,\ \cdots,\ 2\pi'(n) - 1,\quad 2\pi'(1),\ \cdots,\ 2\pi'(n)\big\rangle$$

The two halves interleave perfectly, giving an interleave at the root equal to n. The order in each half of π is the same as π'. Thus, by the recursive construction, each node with $2k$ leaves in its subtree has a perfect interleave of k. Summing over all internal nodes, $\sum_v IL(v) = \frac{1}{2} n \log_2 n$. Refer to Figure 1-3 for an example with $n = 16$, and an illustration of the perfect interleave at each node.

The permutation that we have just constructed is the rather famous *bit-reversal permutation*. Subtracting 1 from every index, we get a permutation of the elements $\{0, \ldots, n-1\}$ which is easy to describe: the value $\pi(i)$ is the number formed by reversing the $\lg n$ bits of i. To see this connection, consider the recursive definition of π: the first half of the values (most significant bit of i is zero) are even (least significant bit of $\pi(i)$ is zero); the second half (most significant bit of i is one) are odd (least significant bit of $\pi(i)$ is one). Recursively, all bits of i except the most significant one appear in π' in reverse order.

Duality of upper and lower bounds. An important theme of this thesis is the idea that a good lower bound should be a natural dual of the best upper bound. The standard upper bound for the partial-sums problem is a balanced binary search tree with the array $A[1\,..\,n]$ in its leaves. Every internal node is augmented to store the sum of all leaves in its subtree. An update recomputes all values on the leaf-to-root path, while a query sums left children hanging from the root-to-leaf path.

Thinking from a lower bound perspective, we can ask when an update (a cell write) to some node v is "actually useful." If v is a left child, the value it stores is used for any query that lies in the subtree of its right sibling. A write to v is useful if a query to the sibling's subtree occurs before another update recomputes v's value. In other words, a write

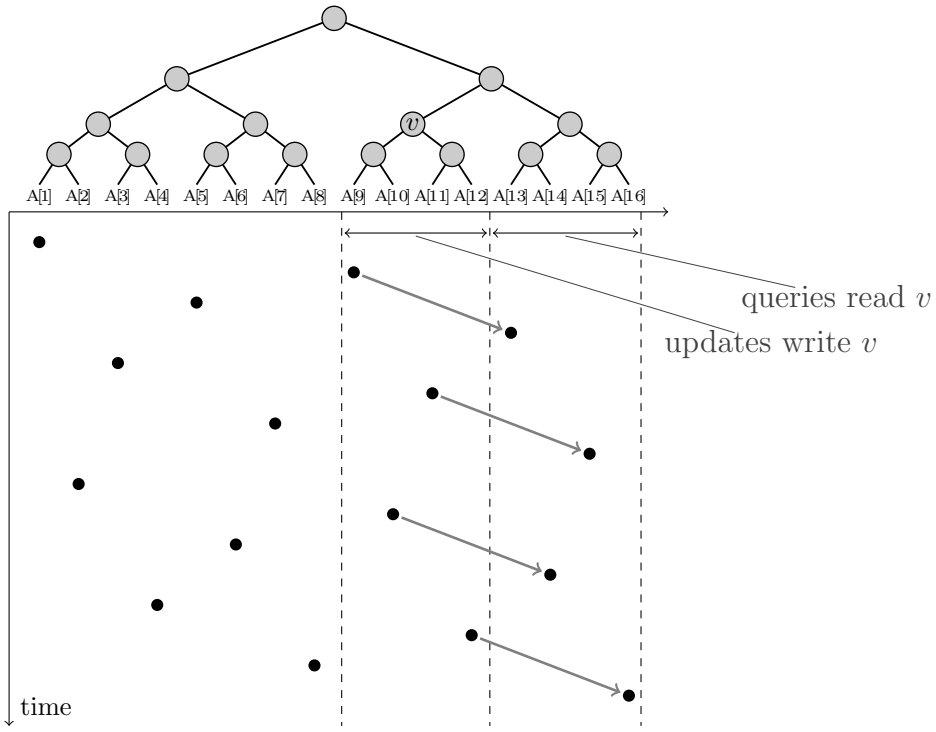

Figure 1-4: An augmented binary search tree solving a partial-sums instance. The writes to some node v are "useful" when interleaves occur in the time-sorted order.

instruction is useful whenever there is an interleave between v's left and right subtrees, sorting operations *by time*. See Figure 1-4.

Intuitively, the amount of "useful work" that the algorithm does is the sum of the interleaves at every node of the binary search tree. To maximize this sum, we can use a bit-reversal permutation again. Note the bit-reversal permutation is equal to its own inverse (reversing the bits twice gives the identity); in other words, the permutation is invariant under 90-degree rotations. Thus, the lower-bound tree sitting on the time axis counts exactly the same interleaves that generate work in the upper bound.

1.6 Dynamic Connectivity: The Hard Instance

We now switch gears to dynamic connectivity. Remember that this problem asks to maintain an undirected graph with a fixed set V of vertices, subject to the following operations:

INSERT(u, v)**:** insert an edge (u, v) into the graph.

DELETE(u, v)**:** delete the edge (u, v) from the graph.

CONNECTED(u, v)**:** test whether u and v lie in the same connected component.

We aim to show an amortized lower bound of $\Omega(\lg |V|)$ per operation.

It turns out that the problem can be dressed up as an instance of the partial sums problem. Let $n = \sqrt{V} - 1$, and consider the partial sums problem over an array $A[1 \,..\, n]$, where each element comes from the group $\mathbb{G} = S_{\sqrt{V}}$, the permutation group on $\sqrt{V}$ elements. We consider a graph whose vertices form an integer grid of size $\sqrt{V}$ by $\sqrt{V}$; see Figure 1-5.

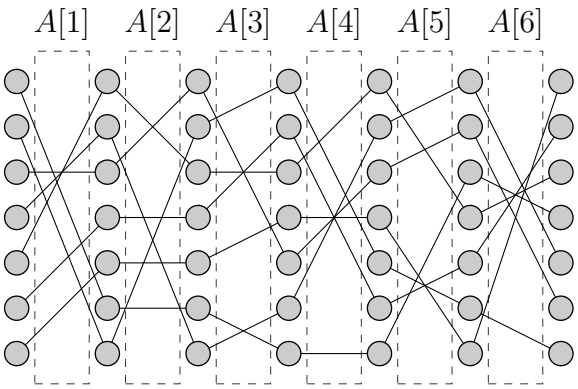

Figure 1-5: Our hard instance of dynamic connectivity implements the partial-sums problem over the group $S_{\sqrt{V}}$.

Edges only connect vertices from adjacent columns; the edges between column i and $i+1$ describe the permutation $A[i]$ from the partial sums problem.

In other words, the graph is a disjoint union of $\sqrt{V}$ paths. Each path stretches from column 1 to column $\sqrt{V}$, and the paths are permuted arbitrarily between columns. This has a strong partial-sums flavor: a node at coordinates $(1, y_1)$ on the first column is connected to a node (k, y_2) on the kth column, if and only if the partial sum permutation $A[1]\circ\cdots\circ A[k-1]$ has y_1 going to y_2.

Given our choice of the group $\mathbb{G}$, we have $\delta = \lg\big((\sqrt{V})!\big) = \Theta\big(\sqrt{V}\cdot\lg V\big)$. For dynamic connectivity, we concentrate on the natural word size $w = \Theta(\lg V)$, so this group is represented by $\Theta(\sqrt{V})$ memory words. Even though these huge group elements may seem worrisome (compared to the previous setting where each $A[i]$ was a word), notice that nothing in our proof depended on the relation between δ and w. Our lower bound still holds, and it implies that a partial-sums operation requires $\Omega(\frac{\delta}{w}\cdot\lg n) = \Omega(\frac{\sqrt{V}\lg V}{\lg V}\cdot\lg\sqrt{V}) = \Omega(\sqrt{V}\cdot\lg V)$ cell probes on average.

Observe that a partial sums UPDATE can be implemented by $O(\sqrt{V})$ dynamic connectivity updates: when some $A[i]$ changes, we run $\sqrt{V}$ DELETE's of the old edges between columns i and $i+1$, followed by $\sqrt{V}$ INSERT's of the new edges. If we could implement SUM using $O(\sqrt{V})$ CONNECTED queries, we would deduce that the dynamic connectivity problem requires a running time of $\Omega(\frac{\sqrt{V}\cdot\lg V}{\sqrt{V}}) = \Omega(\lg V)$ per operation.

Unfortunately, it is not clear how to implement SUM through few CONNECTED queries, since connectivity queries have boolean output, whereas SUM needs to return a permutation with $\Theta\big(\sqrt{V}\cdot\lg V\big)$ bits of entropy. To deal with this issue, we introduce a conceptual change to the partial sums problem, considering a different type of query:
VERIFY-SUM(k, σ)**:** test whether SUM$(k) = \sigma$.

This query is easy to implement via $\sqrt{V}$ connectivity queries: for $i = 1$ to $\sqrt{V}$, these queries test whether the point $(1, i)$ from the first column is connected to point $(k, \sigma(k))$ from the kth column. This runs a pointwise test of the permutation equality $\sigma = A[1]\circ\cdots\circ A[k-1]$.

Below, we extend our lower bound to partial sums with VERIFY-SUM queries:

Theorem 1.8. *In the cell-probe model with w-bit cells, any data structure requires $\Omega(\frac{\delta}{w} n \lg n)$ expected time to support a sequence of n* UPDATE *and n* VERIFY-SUM *operations, drawn from a certain probability distribution.*

By our construction, this immediately implies that, in the cell-probe model with cells of $\Theta(\lg V)$ bits, dynamic connectivity requires $\Omega(\lg V)$ time per operation.

1.7 The Main Trick: Nondeterminism

Our lower bound technique thus far depends crucially on the query answers having high entropy, which lower bounds the information transfer, by Lemma 1.2. High entropy is natural for SUM queries, but impossible for VERIFY-SUM. To deal with this problem, we augment our model of computation with nondeterminism, and argue that SUM and VERIFY-SUM are equivalent in this stronger model.

Technically, a nondeterministic cell-probe algorithm is defined as follows. In the beginning of each query, an arbitrary number of *threads* are created. Each thread proceeds independently according to the following rules:

1. first, the thread may read some cells.
2. the thread decides whether the *accept* or *reject*. Exactly one thread must accept.
3. the accepting thread may now write some cells, and must output the answer.

An alternative view of our model is that an all-powerful prover reveals the query answer, and then probes a minimal set of cells sufficient to certify that the answer is correct. We define the *running time* of the query as the number of cell reads and writes executed by the accepting[1] thread.

A deterministic algorithm for VERIFY-SUM running in time t immediately implies a nondeterministic algorithm for SUM, also running in time t. The algorithm for SUM starts by guessing the correct sum (trying all possibilities in separate threads), and verifying the guess using VERIFY-SUM. If the guess was wrong, the thread rejects; otherwise, it returns the correct answer.

1.8 Proof of the Nondeterministic Bound

We will now show that the lower bound for partial sums holds even for nondeterministic data structures, implying the same bound for VERIFY-SUM queries. The only missing part of the proof is a new version of Lemma 1.2, which bounds the entropy of (nondeterministic) queries in terms of the information transfer.

Remember that in Definition 1.1, we let the information transfer $IT(t_0, t_1, t_2)$ be the set of cells that were: (1) read at a time $t_r \in [t_1, t_2]$; and (2) written at a time $t_w \in [t_0, t_1]$, and

[1]There is no need to consider rejecting threads here. If we have a bound t on the running time of the accepting thread, the algorithm may immediately reject after running for time $t + 1$, since it knows it must be running a rejecting thread.

not overwritten during $[t_w + 1, t_r]$. Condition 2. remains well defined for nondeterministic algorithms, since only the unique accepting thread may write memory cells. For condition 1., we will only look at the reads made by the accepting threads, and ignore the rejecting threads.

More formally, let the *accepting execution* of a problem instance be the sequence of cell reads and writes executed by the updates and the *accepting* threads of each query. Define:

$W(t_0, t_1)$: the set of cells written in the accepting execution during time interval $[t_0, t_1]$.

$R(t_0, t_1)$: the set of cells read in the accepting execution during time interval $[t_0, t_1]$, which had last been written at some time $t_w < t_0$.

Then, $IT(t_0, t_1, t_2) = W(t_0, t_1) \cap R(t_1, t_2)$. Our replacement for Lemma 1.2 states that:

Lemma 1.9. $H\big(A_{[t_1,t_2]} \;\big|\; \Delta^\star = \mathbf{\Delta}^\star\big) \;\le\; O\big(\mathbf{E}\big[w{\cdot}|IT(t_0,t_1,t_2)| + |R(t_0,t_2)| + |W(t_1,t_2)| \;\big|\; \Delta^\star = \mathbf{\Delta}^\star\big]\big)$

Note that this lemma is weaker than the original Lemma 1.2 due to the additional terms depending on $W(t_0, t_1)$ and $R(t_1, t_2)$. However, these terms are fairly small, adding $O(1)$ bits of entropy per cell, as opposed to $O(w)$ bits for each cell in the information transfer. This property will prove crucial.

Before we prove the lemma, we redo the analysis of §1.4, showing that we obtain the same bounds for nondeterministic data structures. As before, we consider a lower-bound tree $\mathcal{T}$, whose n leaves represent time units (query and update pairs). For each internal node v of $\mathcal{T}$, let $[t_0, t_1]$ span the left subtree, and $[t_1, t_2]$ span the right subtree. We then define $IT(v) = |IT(t_0, t_1, t_2)|$, $W(v) = |W(t_0, t_1)|$, and $R(v) = |R(t_1, t_2)|$.

Let T be the total running time of the data structure on a particular instance. As before, observe that each cell read in the accepting execution is counted in exactly one $IT(v)$, at the lowest common ancestor of the read and write times. Thus, $T \ge \sum_{v \in \mathcal{T}} IT(v)$.

For each node, we compare Lemmas 1.9 and 1.4 to obtain a lower bound in terms of the interleave at the node:

$$\mathbf{E}[w \cdot IT(v) + W(v) + R(v)] = \Omega(\delta \cdot IL(v)) \tag{1.1}$$

Note that summing up $R(v) + W(v)$ over the nodes on a single level of the tree gives at most T, because each instruction is counted in at most one node. Thus, summing (1.1) over all $v \in \mathcal{T}$ yields: $\mathbf{E}[w{\cdot}T + T{\cdot}\text{depth}(\mathcal{T})] = \Omega(\delta \sum_v IL(v))$. By using the bit-reversal permutation and letting $\mathcal{T}$ be a perfect binary tree, we have $\sum_v IL(v) = \Omega(n \lg n)$, and $\text{depth}(\mathcal{T}) = \lg n$. Since $w = \Omega(\lg n)$ in our model, the lower bound becomes $\mathbf{E}[2w{\cdot}T] = \Omega(\delta{\cdot}n \lg n)$, as desired.

Proof of Lemma 1.9. The proof is an encoding argument similar to Lemma 1.2, with one additional complication: during decoding, we do not know which thread will accept, and we must simulate all of them. Note, however, that the cells read by the rejecting threads are not included in the information transfer, and thus we cannot afford to include them in the encoding. But without these cells, it is not clear how to decode correctly: when simulating a rejecting thread, we may think incorrectly that a cell was *not* written during $[t_0, t_1]$. If we

give the algorithm a stale version of the cell (from before time t_0), a rejecting thread might now turn into an accepting thread, giving us an incorrect answer.

To fix this, our encoding will contain two components:

C1: for each cell in $IT(t_0, t_1, t_2)$, store the cell address, and the contents at time t_1.

C2: a dictionary for cells in $\big(W(t_0, t_1) \cup R(t_1, t_2)\big) \setminus IT(t_0, t_1, t_2)$, with one bit of associated information: "W" if the cell comes from $W(t_0, t_1)$, and "R" if it comes from $R(t_1, t_2)$.

Component C2 will allow us to stop the execution of rejecting threads that try to read a "dangerous" cell: a cell written in $[t_0, t_1]$, but which is not in the information transfer (and thus, its contents in unknown). The presence of C2 in the encoding accounts for a covert information transfer: the fact that a cell was *not* written during $[t_0, t_1]$ is a type of information that the algorithm can learn during $[t_1, t_2]$.

The immediate concern is that the dictionary of C2 is too large. Indeed, a dictionary storing a set S from some universe U, with some r-bit data associated to each element, requires at least $\lg \binom{|U|}{|S|} + |S| \cdot r$ bits of space. Assuming that the space of cells is $[2^w]$, C2 will use roughly $(|W(t_0, t_1)| + |R(t_1, t_2)|) \cdot w$ bits of space, an unacceptable bound that dominates the information transfer.

We address this concern by pulling an interesting rabbit out of the hat: a retrieval-only dictionary (also known as a "Bloomier filter"). The idea is that the membership (is some $x \in S$?) and retrieval (return the data associated with some $x \in S$) functionalities of a dictionary don't need to be bundled together. If we only need the retrieval query and never run membership tests, we do not actually need to store the set S, and we can avoid the lower bound of $\lg \binom{|U|}{|S|}$ bits:

Lemma 1.10. *Consider a set S from a universe U, where each element of S has r bits of associated data. We can construct a data structure occupying $O(|S| \cdot r + \lg\lg |U|)$ bits of memory that answers the following query:*
RETRIEVE(x) : *if $x \in S$, return x's associated data; if $x \notin S$, return an arbitrary value.*

Proof. To the reader familiar with the field, this is a simple application of perfect hash functions. However, for the sake of completeness, we choose to include a simple proof based on the the probabilistic method.

Let $n = |S|, u = |U|$. Consider a hash function $h : U \to [2n]$. If the function is injective on S, we can use an array with $2n$ locations of r bits each, storing the data associated to each $x \in S$ at $h(x)$. For retrieval, injectivity of S guarantees that the answer is correct whenever $x \in S$.

There are $\binom{2n}{n} \cdot n! \cdot (2n)^{u-n}$ choices of h that are injective on S, out of $(2n)^u$ possibilities. Thus, if h is chosen uniformly at random, it works for any fixed S with probability $\binom{2n}{n} \cdot n!/(2n)^n \geq 2^{-O(n)}$. Pick a family $\mathcal{H}$ of $2^{O(n)} \cdot \lg \binom{u}{n}$ independently random h. For any fixed S, the probability that no $h \in \mathcal{H}$ is injective on S is $(1 - \frac{1}{2^{O(n)}})^{|\mathcal{H}|} = \exp\big(\Theta\big(\lg \binom{u}{n}\big)\big) < 1/\binom{u}{n}$. By a union bound over all $\binom{u}{n}$ choices of S, there exists a family $\mathcal{H}$ such that for any S, there exists $h \in \mathcal{H}$ injective on S.

Since our lemma does not promise anything about the time efficiency of the dictionary, we can simply construct $\mathcal{H}$ by iterating over all possibilities. The space will be $2n \cdot r$ bits for

the array of values, plus $\lg|\mathcal{H}| = O(n + \lg\lg u)$ bits to specify a hash function from $\mathcal{H}$ that is injective on S. □

We will implement C2 using a retrieval-only dictionary, requiring $O\big(|W(t_0,t_1)|+|R(t_1,t_2)|\big)$ bits of space. Component C1 requires $O(w)\cdot|IT(t_0,t_1,t_2)|$ bits. It only remains to show that the query answers $A_{[t_1,t_2]}$ can be recovered from this encoding, thus giving an upper bound on the entropy of the answers.

To recover the answers $A_{[t_1,t_2]}$, we simulate the execution of the data structure during $[t_1,t_2]$. Updates, which do not use nondeterminism, are simulated as in Lemma 1.2. For a query happening at time $t \in [t_1,t_2]$, we simulate all possible threads. A cell read by one of these threads falls into one of the following cases:

$W(t_1,t)$: We can recognize this case by maintaining a list of memory locations written during the simulation; the contents of the cell is immediately available.

$IT(t_0,t_1,t_2)$: We can recognize this case by examining the addresses in C1; the cell contents can be read from the encoding.

$W(t_0,t_1)\setminus IT(t_0,t_1,t_2)$: We can recognize this case by querying the dictionary C2. If the retrieval query returns "W," we know that the answer *cannot* be "R" (the correct answer may be "W," or the cell may be outside the dictionary set, in which case an arbitrary value is returned). But if the answer cannot be "R," the cell cannot be in $R(t_1,t_2)$. Thus, this thread is certainly not the accepting thread, and the simulation may reject immediately.

$W(1,t_0-1)\setminus W(t_0,t)$: This is the default case, if the cell doesn't satisfy previous conditions. The contents of the cell is known, because the operations before time t_0 are fixed, as part of $\Delta^\star$.

It should be noted that C2 allows us to handle an arbitrary number of rejecting threads. All such threads are either simulated correctly until they reject, or the simulation rejects earlier, when the algorithm tries to read a cell in $W(t_0,t_1)\setminus IT(t_0,t_1,t_2)$.

1.9 Bibliographical Notes

In our paper [PD06], we generalize the argument presented here to prove lower bound trade-offs between the update time t_u and the query time t_q. We omit these proofs from the current thesis, since our improved epoch arguments from the next chapter will yield slightly better trade-offs than the ones obtained in [PD06].

Dictionaries supporting only retrieval have found another beautiful application to the range reporting problem in one dimension. See our paper [MPP05] for the most recent work on 1-dimensional range reporting. *Dynamic* dictionaries with retrieval were investigated in our paper [DadHPP06], which gives tight upper and lower bounds.