Chapter 20

Cost of Sequences

In this chapter, we present several cost specifications for the Sequence ADT Chapter . These cost specifications pertain to implementations that use several common representations for sequences based on arrays , trees , and lists .

The cost specifications do not require describing the particular implementations, but implementations that match the given costs indeed use data structures based on arrays, trees, and lists (respectively). However, for example, there might many tree implementations that match the tree cost specification.

All of the the cost bounds we give are based on "pure" implementations, as discussed in Functional Algorithms Chapter . In particular all functions create new data without changing the old data. In the case of the array cost specification, this means that updates are expensive since they need to copy the array. In Ephemeral Sequences we discuss both the "inpure" costs and an interface that remains pure, but reduces the cost to the same as the "impure" case when used in a particular way.

1 Cost Specifications

Cost specifications describe the cost—in terms of work and span—of the functions in an ADT. Typically many specific implementations match a specific cost specification. For example, for the tree-sequence specification for sequences (Section 3, an implementation can use one of many balanced binary tree data structures available.

To use a cost specification, we don't need to know the details of how these implementations work. Cost specifications can thus be viewed as an abstraction over implementation details that do not matter for the purposes of the algorithm.

Note. Cost specifications are similar to prices on restaurant menus. If we view the functions of the ADT as the dishes in a menu, then the cost specification is the price tag for each dish. Just as the cost of the dishes in a menu does not change from day to day as the specific details of the preparation process changes (e.g., different cooks may prepare the dish, the

origin of the ingredients may vary from one day to the next), cost specifications offer a layer of abstraction over implementation details that a client of the ADT need not know.

Definition 20.1 (Domination of cost specifications)**.** There are usually multiple ways to implement an ADT and thus there can be multiple cost specifications for the same ADT. We say that one cost specification ***dominates*** another if for each and every function, its asymptotic costs are no higher than those of the latter.

Example 20.1. Of the three cost specifications that we consider in this chapter, none dominates another. The list-based cost specification, however, is almost dominated by the others, because it is nearly completely sequential.

Choosing cost specifications. When deciding which of the possibly many cost specification to use for a particular ADT, we usually notice that there are certain trade-offs: some functions will be cheaper in one and while others are cheaper in another. In such cases, we choose the cost specification that minimizes the cost for the algorithm that we wish to analyze. After we decide the specification to use, what remains is to select the implementation that matches the specification, which can include additional considerations.

Example 20.2. If an algorithm makes many calls to nth but no calls to $append$, then we would use the array-sequence specification rather than the tree-sequence specification, because in the former nth requires constant work whereas in the latter it requires logarithmic work. Conversely, if the algorithm mostly uses $append$ and $update$, then tree-sequence specification would be better.

Note. Following on our restaurant analogy, suppose that you wish to enjoy a nice three course meal in a nearby restaurant. Looking over the menues of the restaurants in your price range, you might realize that prices for the appetizers in one are lower than others but the main dishes are more expensive. (If that were not the case, the others would be dominated and, assuming equal quality, they would likely go out of business.) Assuming your goal is to minimize the total cost of your meal, you would therefore sum up the cost for the dishes that you plan on enjoying and make your decision based on the total sum.

2 Array Sequences

Cost Specification 20.2 (Array Sequences)**.** The table below specifies the ***array-sequence*** costs. For the cost of $inject$, we define the degree of an update sequence as the maximum number of updates targeting the same position. The notation $\mathcal{T}(-)$ refer to the trace of the corresponding operation. The specification for $scan$ assumes that f has constant work and span.

Operation	Work	Span
length a	1	1
nth a i	1	1
singleton x	1	1
empty	1	1
isSingleton x	1	1
isEmpty x	1	1
tabulate f n	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \max_{i=0}^{n} S(f(i))$
map f a	$1 + \sum_{x \in a} W(f(x))$	$1 + \max_{x \in a} S(f(x))$
filter f a	$1 + \sum_{x \in a} W(f(x))$	$\lg\|a\| + \max_{x \in a} S(f(x))$
subseq a (i, j)	1	1
append a b	$1 + \|a\| + \|b\|$	1
flatten a	$1 + \|a\| + \sum_{x \in a} \|x\|$	$1 + \lg\|a\|$
update a (i, x)	$1 + \|a\|$	1
inject a b	$1 + \|a\| + \|b\|$	$\lg(\mathsf{degree}(b))$
ninject a b	$1 + \|a\| + \|b\|$	1
collect f a	$1 + W(f) \cdot \|a\| \lg\|a\|$	$1 + S(f) \cdot \lg^2\|a\|$
iterate f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
reduce f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$\lg\|a\| \cdot \max_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
scan f x a	$\|a\|$	$\lg\|a\|$

The functions *length*, *nth*, *empty*, *singleton*, *isEmpty*, *isSingleton*, *subseq*, all require constant work and span.

Tabulate and Map. The functions *tabulate* and *map* total work that is equal to the sum of the work of applying f at each position, as well as an additional unit cost to account for *tabulate* or *map* itself.

Because it is possible to apply the function f in parallel—there are no dependencies among the different positions, the span is the maximum of the span of applying f at each position, plus 1 for the function call itself.

Example 20.3 (Tabulate and map with array costs)**.** As an example of *tabulate* and *map*

$$\begin{array}{rcccl} W\left(\left\langle\, i^2 : 0 \leq i < n \,\right\rangle\right) & = & O\left(1 + \sum_{0=1}^{n-1} O(1)\right) & = & O(n) \\ S\left(\left\langle\, i^2 : 0 \leq i < n \,\right\rangle\right) & = & O\left(1 + \max_{i=0}^{n-1} O(1)\right) & = & O(1) \end{array}$$

because the work and span for i^2 is $O(1)$.

Filter. The work for the function *filter* is equal to the sum of the work of applying f at each position, as well as an additional unit cost, for the function call itself.

Because it is possible to apply the function f in parallel—there are no dependencies among the different positions, the span is the maximum of the span of applying f at each position, plus a logarithmic term for performing ***compaction***, i.e., packing the chosen elements contiguously into the result array.

Example 20.4 (Filter)**.** As an example of *filter*, we have

$$\begin{array}{rcll} W\left(\langle\, x : x \in a \mid x < 27 \,\rangle\right) & = & O\left(1 + \sum_{i=0}^{|a|-1} O\left(1\right)\right) & = & O\left(|a|\right) \\ S\left(\langle\, x : x \in a \mid x < 27 \,\rangle\right) & = & O\left(\lg|a| + \max_{i=0}^{|a|-1} O\left(1\right)\right) & = & O(\lg|a|). \end{array}$$

The operation *append* requires work proportional to the length of the sequences given as input, can be implemented in constant span.

The operation *flatten* generalizes *append*, requiring work proportional to the total length of the sequences flattened, and can be implemented in parallel in logarithmic span in the number of sequences flattened.

Update and Inject. The operations *update* and *inject* both require work proportional to the length of the sequences they are given as input. It might seem surprising that *update* takes work proportional to the size of the input sequence a, since updating a single element should require constant work. The reason is that the interface is purely functional so that the input sequence needs to be copied–we are not allowed to update the old copy.

The function *update* and non-deterministic inject *ninject* can be implemented in constant span, but deterministic inject required resolving conflicts more carefully and requires $O\left(\lg(\mathsf{degree}(b))\right)$ span where the degree of the update sequence b is the maximum number of updates targeting the same position in the sequence being updated.

In the last section of this chapter, we describe single-threaded array sequences that allows updating under a sequence in constant work, but under certain restrictions.

Collect. The primary cost in implementing *collect* is a sorting step that sorts the sequence based on the keys. The work and span of collect is therefore determined by the work and span of (comparison) sorting with the specified comparison function f.

Cost of aggregation. The cost of aggregation functions, *iterate*, *reduce*, and *scan* are more difficult to specify, because they depend their arguments and on the intermediate values computed during evaluation.

Example 20.5 (Cost of Iterated *append*)**.** Consider appending the following sequence of strings using *iterate*:

iterate append `''` $\langle$`'abc','d','e','f'`$\rangle$.

If we only count the work of *append* functions performed during evaluation, we obtain a total work of 22, because the following *append* functions are performed

1. *append* `'''abc'` (work 4),
2. *append* `'abc''d'` (work 5),
3. *append* `'abcd''e'` (work 6), and
4. *append* `'abcde''f'` (work 7).

Consider now appending the following sequence of strings, which is a permutation of the previous, using *iterate*:

$$\mathit{iterate}\ \mathit{append}\ \texttt{''}\ \langle \texttt{'d'}, \texttt{'e'}, \texttt{'f'}, \texttt{'abc'} \rangle$$

If we only count the work of *append* operations using the array-sequence specification, we obtain a total work of 16, because the following *append* operations are performed

1. *append* `'''d'` (work 2),
2. *append* `'d''e'` (work 3),
3. *append* `'de''f'`, (work 4) and
4. *append* `'def''abc'` (work 7).

Thus, we have used iteration over two sequences, both with 4 elements, and obtained different costs even though the sequences are permutations of each other. The reason for this is that the total cost depends on the intermediate values generated during computation.

Specification of *iterate*. To specify the cost of *iterate*, we consider the intermediate values generated by an evaluation of *iterate*, whose specification, originally given in Section 9 is reproduced here for convenience.

$$\mathit{iterate}\ f\ x\ a = \begin{cases} x & \text{if } |a| = 0 \\ \mathit{iterate}\ f\ (f(x, a[0]))\ (a[1 \cdots |a| - 1]) & \text{otherwise.} \end{cases}$$

Consider an evaluation of

$$\mathit{iterate}\ f\ v\ a$$

and let

$$\mathcal{T}(\mathit{iterate}\ f\ v\ a)$$

denote the set of calls to $f(\cdot, \cdot)$ performed along with the arguments, as defined by the specification above. We refer to this set of function calls as the ***trace*** of *iterate* and define the cost of *iterate* as the sum of these calls.

Cost Specification 20.3 (Cost for $iterate$)**.** Consider evaluation of $iterate\ f\ v\ a$ and let $\mathcal{T}(iterate\ f\ v\ a)$ denote the set of calls (trace) to $f(\cdot,\cdot)$ performed along with the arguments. The work and span are as follows.

$$\begin{array}{lcl} W\left(iterate\ f\ x\ a\right) & = & O\left(1+\sum_{f(y,z)\in\mathcal{T}(iterate\ f\ x\ a)} W\left(f(y,z)\right)\right) \\ S\left(iterate\ f\ x\ a\right) & = & O\left(1+\sum_{f(y,z)\in\mathcal{T}(iterate\ f\ x\ a)} S\left(f(y,z)\right)\right) \end{array}$$

Example 20.6 (Sorting by Iteration)**.** As an interesting example, consider the function $mergeOne\ a\ x$ for merging a sequence a with the singleton sequence $\langle\, x\,\rangle$ by using an assumed comparison function. The function performs $O(n)$ work in $O(\lg n)$ span, where n is the total number of elements in the output sequence. We can use the $mergeOne$ function to sort a sequence via iteration as follows

$$iterSort\ a = iterate\ mergeOne\ \langle\,\rangle\ a.$$

For example, on input $a = \langle\, 2, 1, 0\,\rangle$, $iterSort$ first merges $\langle\,\rangle$ and $\langle\, 2\,\rangle$, then merges the result $\langle\, 2\,\rangle$ with $\langle\, 1\,\rangle$, then merges the resulting sequence $\langle\, 1, 2\,\rangle$ with $\langle\, 0\,\rangle$ to obtain the final result $\langle\, 0, 1, 2\,\rangle$.

The trace for $iterSort$ with an input sequence of length n consists of a set of calls to $mergeOne$, where the first argument is a sequence of sizes varying from 1 to $n-1$, while its right argument is always a singleton sequence. For example, the final $mergeOne$ merges the first $(n-1)$ elements with the last element, the second to last $mergeOne$ merges the first $(n-2)$ elements with the second to last element, and so on. Therefore, the total work for an input sequence a of length n is

$$W\left(iterSort\ a\right) \leq \sum_{i=1}^{n-1} c \cdot (1+i) = O(n^2).$$

Using the trace, we can also analyze the span of $iterSort$. Since we iterate adding in each element after the previous, there is no parallelism between merges, but there is parallelism within a $mergeOne$, whose span is is logarithmic. We can calculate the total span as

$$S\left(iterSort\ a\right) \leq \sum_{i=1}^{n-1} c \cdot \lg\left(1+i\right) = O(n \lg n).$$

Since average parallelism, $W\left(n\right)/S\left(n\right) = O(n/\lg n)$, we see that the algorithm has a reasonable amount of parallelism. Unfortunately, it does much too much work.

Note (Algorithm $iterSort$). Using this reduction order the algorithm is effectively working from the front to the rear, using $mergeOne$ to "insert" each element into a sorted prefix where it is placed at the correct location to maintain the sorted order. The algorithm thus implements the well-known insertion sort.

Cost of $reduce$**.** Recall that with $reduce$, we noted that the result of the computation is not affected by the order in which the associative function is applied and in fact is the same as that of performing the same computation with $iterate$. The cost of $reduce$, however, depends on the order in which the operations are performed.

Example 20.7 (Cost of reduce append)**.** Consider appending the following code

$$\mathit{reduce}\ \mathit{append}\ \texttt{''}\ \langle \texttt{'abc'},\texttt{'d'},\texttt{'e'},\texttt{'f'}\rangle\,.$$

Suppose performing append operations in left-to-right order and count their work using the array-sequence specification. The total work is 19, because the following *append* operations are performed

1. *append*' `abc` '' `d` ' (work 5),
2. *append*' `abcd` '' `e` ' (work 6), and
3. *append*' `abcde` '' `f` ' (work 7).

Consider now performing the *append* operations from right to left order. We obtain a total cost of 15, because the following *append* operations are performed

1. *append* `'e''f'` (work 3),
2. *append* `'d''ef'`, (work 4) and
3. *append* `'abc''def'` (work 7).

Specification of *reduce*. To specify the cost of reduce, we consider its trace based on its specification, as given in Section 10 reproduced below for convenience.

$$\mathit{reduce}\ f\ \mathit{id}\ a = \begin{cases} \mathit{id} & \text{if } |a| = 0 \\ a[0] & \text{if } |a| = 1 \\ f\Big(\mathit{reduce}\ f\ \mathit{id}\ (a[0 \cdots \lfloor \frac{|a|}{2} \rfloor - 1]), & \\ \quad \mathit{reduce}\ f\ \mathit{id}\ (a[\lfloor \frac{|a|}{2} \rfloor \cdots |a| - 1]\Big) & \text{otherwise.} \end{cases}$$

Cost Specification 20.4 (Cost of *reduce*)**.** Consider evaluation of $\mathit{reduce}\ f\ x\ a$ and let $\mathcal{T}(\mathit{reduce}\ f\ x\ a)$ denote the set of calls to $f(\cdot,\cdot)$ performed along with the arguments. The work and span are defined as

$$\begin{aligned} W\,(\mathit{reduce}\ f\ x\ a) &= O\left(1 + \sum_{f(y,z)\in\mathcal{T}(\mathit{reduce}\ f\ x\ a)} W\,(f(y,z))\right), \text{ and} \\ S\,(\mathit{reduce}\ f\ x\ a) &= O\left(\lg|a| \cdot \max_{f(y,z)\in\mathcal{T}(\mathit{reduce}\ f\ x\ a)} S\,(f(y,z))\right). \end{aligned}$$

Work and Span of reduce. The work bound is simply the total work performed, which we obtain by summing across all combine functions, plus one for the *reduce*. The span bound is more interesting. The $\lg|a|$ term expresses the fact that the recursion tree in the specification of *reduce* is at most $O(\lg|a|)$ deep. Since each node in the recursion tree has span at most $\max_{f(y,z)} S\,(f(y,z))$, any root-to-leaf path, has at most $O(\lg|a| \cdot \max_{f(a,b)} S\,(f(a,b)))$ span.

Cost of $scan$**.** As in $iterate$ and $reduce$ the cost specification of $scan$ depends on the intermediate results. But the dependency is more complex than can be represented by our ADT specification. For $scan$, we will stop at giving a cost specification by assuming that the function that we are scanning with performs $O(1)$ work and span.

Cost Specification 20.5 (Cost for $scan$)**.** Consider the expression $scan\ f\ x\ a$, where $f(\cdot,\cdot)$ always requires $O(1)$ work and span. The work and span of the expression are defined as

$$\begin{array}{lcl} W(scan\ f\ x\ a) & = & O(|a|), \text{ and} \\ S(scan\ f\ x\ a) & = & O(\lg|a|). \end{array}$$

3 Tree Sequences

The costs for tree sequences is given in Cost Specification below . The specification represents the cost for a class of implementations that use a balanced tree to represent the sequence. The cost of each operation is similar to the array-based specification, and many are exactly the same, i.e., $length$, $singleton$, $isSingleton$, $isEmpty$, $collect$, $iterate$, $reduce$, and $scan$.

There are also differences. The work and span of the operation nth is logarithmic, as opposed to being constant. This is because in balanced-tree based implementation, the operation must follow a path from the root to a leaf to find the desired element element. For a sequence a, such a path has length $O(\lg|a|)$. Although nth does more work with tree sequences, $append$ does less work. Instead of requiring linear work, the work of $append$ with tree sequences is proportional to the logarithm of the ratio of the size of the larger sequence to the size of the smaller one smaller one. For example if the two sequences are the same size, then $append$ takes $O(1)$ work. On the other hand if one is length n and the other 1, then the work is $O(\lg n)$. The work of $update$ is also less with tree sequences than with array sequences.

The work for operations map and $tabulate$ are the same as those for array sequences; their span incurs an extra logarithmic overhead. The work and span of $filter$ are the same for both.

Cost Specification 20.6 (Tree Sequences)**.** We specify the ***tree-sequence*** costs as follows. The notation $\mathcal{T}(-)$ refer to the trace of the corresponding operation. The specification for $scan$ assumes that f has constant work and span.

Operation	Work	Span
length a	1	1
singleton x	1	1
isSingleton x	1	1
isEmpty x	1	1
nth a i	$\lg\|a\|$	$\lg\|a\|$
tabulate f n	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \lg n + \max_{i=0}^{n} S(f(i))$
map f a	$1 + \sum_{x \in a} W(f(x))$	$1 + \lg\|a\| + \max_{x \in a} S(f(x))$
filter f a	$1 + \sum_{x \in a} W(f(x))$	$1 + \lg\|a\| + \max_{x \in a} S(f(x))$
subseq(a, i, j)	$1 + \lg(\|a\|)$	$1 + \lg(\|a\|)$
append a b	$1 + \|\lg(\|a\|/\|b\|)\|$	$1 + \|\lg(\|a\|/\|b\|)\|$
flatten a	$1 + \|a\| \lg\left(\sum_{x \in a} \|x\|\right)$	$1 + \lg(\|a\| + \sum_{x \in a} \|x\|)$
inject a b	$1 + (\|a\| + \|b\|) \lg\|a\|$	$1 + \lg(\|a\| + \|b\|)$
ninject a b	$1 + (\|a\| + \|b\|) \lg\|a\|$	$1 + \lg(\|a\| + \|b\|)$
collect f a	$1 + W(f) \cdot \|a\| \lg\|a\|$	$1 + S(f) \cdot \lg^2\|a\|$
iterate f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
reduce f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$\lg\|a\| \cdot \max_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
scan f x a	$\|a\|$	$\lg\|a\|$

4 List Sequences

The Cost Specification below defines the cost for list sequences. The specification represents the cost for a class of implementations that use (linked) lists to represent the sequence. The determining cost in list-based implementations is the sequential nature of the representation: accessing the element at position i requires traversing the list from the head to i, which leads to $O(i)$ work and span. List-based implementations therefore expose hardly any parallelism. Their main advantage is that they require quick access to the ***head*** and the ***tail*** of the sequence, which are defined as the first element and the suffix of the sequence that starts at the second element respectively.

The work of each operation is similar to the array-based specification. Since the data structure mostly serial, the span of each operation is essentially the same as that of its work, except that the total is taken over the spans of its components. The work and span of *subseq* operation depends on the beginning position of the subsequence, because list-based representation can share their suffixes.

Cost Specification 20.7 (List Sequences)**.** We specify the ***list-sequence*** costs as follows. The notation $\mathcal{T}(-)$ refer to the trace of the corresponding operation. The specification for *scan* assumes that f has constant work and span.

Operation	Work	Span
length a	1	1
singleton x	1	1
isSingleton x	1	1
isEmpty x	1	1
nth a i	i	i
tabulate f n	$1 + \sum_{i=0}^{n} W(f(i))$	$1 + \sum_{i=0}^{n} S(f(i))$
map f a	$1 + \sum_{x \in a} W(f(x))$	$1 + \sum_{x \in a} S(f(x))$
filter f a	$1 + \sum_{x \in a} W(p(x))$	$1 + \sum_{x \in a} S(p(x))$
subseq a (i, j)	$1 + i$	$1 + i$
append a b	$1 + \lvert a \rvert$	$1 + \lvert a \rvert$
flatten a	$1 + \lvert a \rvert + \sum_{x \in a} \lvert x \rvert$	$1 + \lvert a \rvert + \sum_{x \in a} \lvert x \rvert$
update a (i, x)	$1 + \lvert a \rvert$	$1 + \lvert a \rvert$
inject a b	$1 + \lvert a \rvert + \lvert b \rvert$	$1 + \lvert a \rvert + \lvert b \rvert$
ninject a b	$1 + \lvert a \rvert + \lvert b \rvert$	$1 + \lvert a \rvert + \lvert b \rvert$
collect f a	$1 + W(f) \cdot \lvert a \rvert \lg \lvert a \rvert$	$1 + S(f) \cdot \lvert a \rvert \lg \lvert a \rvert$
iterate f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
reduce f x a	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} W(f(y,z))$	$1 + \sum_{f(y,z) \in \mathcal{T}(-)} S(f(y,z))$
scan f a	$\lvert a \rvert$	$\lvert a \rvert$

Remark. Since they are serial, list-based sequences are usually ineffective for parallel algorithm design.