

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

Parallelism vs. Concurrency

Parallelism and concurrency sound like synonyms, but they are **not** directly related to each other at all.

Parallelism vs. Concurrency

Parallelism and concurrency sound like synonyms, but they are **not** directly related to each other at all.

Definition

Parallelism is the simultaneous execution of code on multiple processors or cores for the purposes of improved **performance**.

Example Applications: Simulations (weather, n-body, fluid flow), Stencil Convolutions, Neural Networks.

Parallelism vs. Concurrency

Parallelism and concurrency sound like synonyms, but they are **not** directly related to each other at all.

Definition

Parallelism is the simultaneous execution of code on multiple processors or cores for the purposes of improved **performance**.

Example Applications: Simulations (weather, n-body, fluid flow), Stencil Convolutions, Neural Networks.

Definition

Concurrency is an abstraction for the programmer, allowing programs to be structured as multiple **threads of control**, called **processes**. These processes may communicate in various ways.

Example Applications: Servers, OS Kernels, GUI applications.

Parallelism vs. Concurrency

While any parallel algorithm ultimately must involve concurrency at the level of processor instructions, this is not necessarily the case at the language level.

- **Parallelism without Concurrency:** Parallel languages like TensorFlow, Accelerate (Haskell), Futhark, Parallel Fortran, CUDA.
- **Concurrency without Parallelism:** Any concurrent program running on a single core processor or a kernel without multiprocessing support.

While it is **possible** to describe parallel algorithms using concurrency abstractions, this is **not usually desirable**.

Stay away from Concurrency¹

Concurrent programs are highly *non-deterministic*. The order in which each process takes steps is up to the scheduler of the operating system.

This necessitates a means of *synchronizing* processes so that they can produce predictable results. Failure to do so correctly results in *heisenbugs*.

¹for Parallelism

Stay away from Concurrency¹

Concurrent programs are highly *non-deterministic*. The order in which each process takes steps is up to the scheduler of the operating system.

This necessitates a means of *synchronizing* processes so that they can produce predictable results. Failure to do so correctly results in *heisenbugs*.

The complexity of reasoning about concurrent systems grows (in the worst case) *exponentially with the number of processes*.

In many parallel algorithms, the number of processes is *unbounded*: reasoning about thousands of processes using concurrency models gets intractable very easily.

¹for Parallelism

Parallelism Structure

There are two main types of parallelism:

- 1 **Data Parallelism** Also called *single instruction, multiple data* (SIMD). The same code is executed on multiple processing units, working on different parts of a large data set.

Parallelism Structure

There are two main types of parallelism:

- 1 **Data Parallelism** Also called *single instruction, multiple data* (SIMD). The same code is executed on multiple processing units, working on different parts of a large data set.

Examples: GPUs

Advantages: Simple, Scalable

Disadvantages: Maybe limited applications.

Parallelism Structure

There are two main types of parallelism:

- 1 **Data Parallelism** Also called *single instruction, multiple data* (SIMD). The same code is executed on multiple processing units, working on different parts of a large data set.
Examples: GPUs
Advantages: Simple, Scalable
Disadvantages: Maybe limited applications.
- 2 **Control Parallelism** Also called *task parallelism* or *multiple instruction, multiple data* (MIMD). Different programs are executed on different processing units.

Parallelism Structure

There are two main types of parallelism:

- 1 **Data Parallelism** Also called *single instruction, multiple data* (SIMD). The same code is executed on multiple processing units, working on different parts of a large data set.

Examples: GPUs

Advantages: Simple, Scalable

Disadvantages: Maybe limited applications.

- 2 **Control Parallelism** Also called *task parallelism* or *multiple instruction, multiple data* (MIMD). Different programs are executed on different processing units.

Examples: Multicore CPUs

Advantages: Flexible, Widely-supported

Disadvantages: Less scalable, explicit synchronization.

SIMD and MIMD

- We can compile SIMD code to MIMD code straightforwardly, but typically synchronization code must be generated, which is hard to do optimally.

SIMD and MIMD

- We can compile SIMD code to MIMD code straightforwardly, but typically synchronization code must be generated, which is hard to do optimally.
- We can compile MIMD code to SIMD code, by branching on the current process ID, however this leads to **atrocious** performance on SIMD hardware.

Determinism and Scalability

Ideally, we should be able to describe a parallel algorithm in our language, and expect **deterministic** results. That is, no matter how many processors or cores are used, we should get the same result.

Ideally, performance should improve linearly with the number of processing units we supply. This is called *scalability*.

Control parallelism is difficult to make deterministic², and impossible to make generally scalable. For that reason, we'll focus on **data parallelism**.

²Although possible, see Haskell's spark model.

Data Parallel Languages

Key Idea

We will introduce a *parallel array* type to MinHS, along with a series of collective operations to operate on arrays in parallel.

τ	::=	...	
		$[\tau]$	(parallel arrays)
e	::=	...	
		$[e_1, e_2, \dots, e_n]$	(constant arrays)
		$e_1[e_2]$	(array indexing)
		$\text{len}(e)$	(length)
		$[e_r \mid x \leftarrow e_1 \mid \dots \mid x \leftarrow e_n]$	(zip comprehensions)
		$[e_r \mid x \leftarrow e, e_g]$	(filter comprehensions)
		$e_1 ++ e_2$	(concatenation)
		$\text{fold}_{\otimes} e$	(reduction)

Operational Intuitions

Let $xs = [[1, 2], [], [3, 4, 5, 6]]$. Then:

- xs has type

Operational Intuitions

Let $xs = [[1, 2], [], [3, 4, 5, 6]]$. Then:

- xs has type $[[Int]]$ – an array of arrays of integers.
- $xs[2]$

Operational Intuitions

Let $xs = [[1, 2], [], [3, 4, 5, 6]]$. Then:

- xs has type `[[Int]]` – an array of arrays of integers.
- $xs[2]$ is standard array indexing, 0-based, so we get `[3, 4, 5, 6]`.
- `len(xs)`

Operational Intuitions

Let $xs = [[1, 2], [], [3, 4, 5, 6]]$. Then:

- xs has type $[[Int]]$ – an array of arrays of integers.
- $xs[2]$ is standard array indexing, 0-based, so we get $[3, 4, 5, 6]$.
- $len(xs)$ returns 3.
- $[x + y \mid x \leftarrow [1, 2, 3, 4] \mid y \leftarrow [5, 6, 7, 8]]$

Operational Intuitions

Let $xs = [[1, 2], [], [3, 4, 5, 6]]$. Then:

- xs has type $[[Int]]$ – an array of arrays of integers.
- $xs[2]$ is standard array indexing, 0-based, so we get $[3, 4, 5, 6]$.
- $len(xs)$ returns 3.
- $[x + y \mid x \leftarrow [1, 2, 3, 4] \mid y \leftarrow [5, 6, 7, 8]]$ evaluates to $[6, 8, 10, 12]$ **in parallel**.
- $[1 + x \mid x \leftarrow [1, 2, 3, 4], x \bmod 2 == 0]$

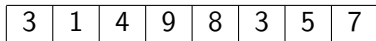
Operational Intuitions

Let $xs = [[1, 2], [], [3, 4, 5, 6]]$. Then:

- xs has type `[[Int]]` – an array of arrays of integers.
- $xs[2]$ is standard array indexing, 0-based, so we get `[3, 4, 5, 6]`.
- `len(xs)` returns 3.
- `[x + y | x ← [1, 2, 3, 4] | y ← [5, 6, 7, 8]]` evaluates to `[6, 8, 10, 12]` **in parallel**.
- `[1 + x | x ← [1, 2, 3, 4], x mod 2 == 0]` evaluates to `[3, 5]` **in parallel**.

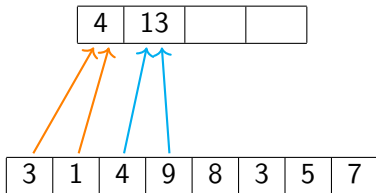
Reductions

How do we evaluate fold? Take fold_+ as an example, assuming we have two cores:



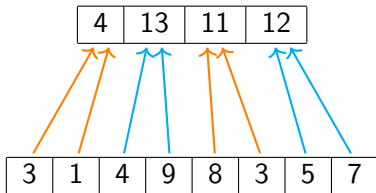
Reductions

How do we evaluate fold? Take fold_+ as an example, assuming we have two cores:



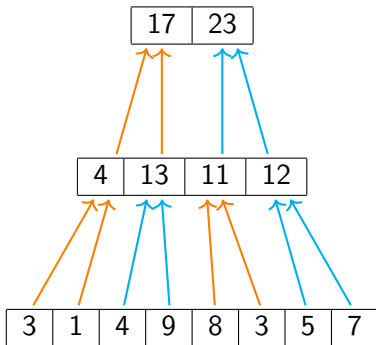
Reductions

How do we evaluate fold? Take fold_+ as an example, assuming we have two cores:



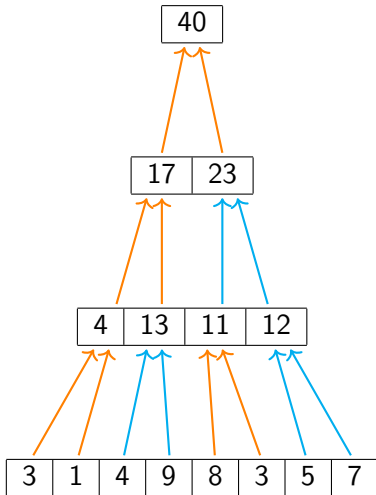
Reductions

How do we evaluate fold? Take fold_+ as an example, assuming we have two cores:



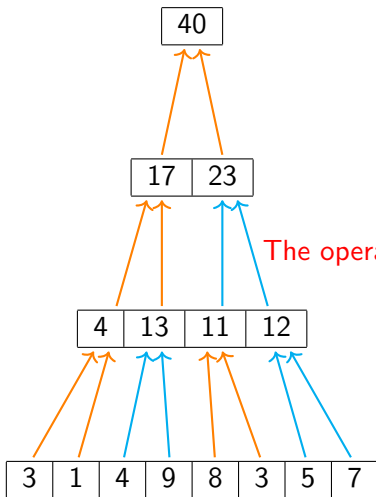
Reductions

How do we evaluate fold? Take fold_+ as an example, assuming we have two cores:



Reductions

How do we evaluate fold? Take fold_+ as an example, assuming we have two cores:



The operator must be associative for determinism!

Examples

Example (Vector Dot Product)

```
recfun dotProduct :: ([Float] → [Float] → Float) xs ys =  
  fold+ [x + y | x ← xs | y ← ys]
```

Examples

Example (Vector Dot Product)

```
recfun dotProduct :: ([Float] → [Float] → Float) xs ys =  
  fold+ [x + y | x ← xs | y ← ys]
```

Example (Sparse Matrix-Vector Multiplication)

```
type SparseVector = [Int × Float]  
type SparseMatrix = [SparseVector]  
recfun smvm :: (SparseMatrix → [Float] → [Float]) sm v =  
  [fold+ [ v[fst elem] × snd elem | elem ← sv ] | sv ← sm]
```

Flat vs. Nested Data Parallelism

The second example, *smvm*, performed a **nested parallel loop**, i.e. a parallel computation inside another parallel computation. This is called *nested data parallelism*.

Flat vs. Nested Data Parallelism

The second example, *smvm*, performed a **nested parallel loop**, i.e. a parallel computation inside another parallel computation. This is called *nested data parallelism*.

Most data-parallel architectures do **not** support such things directly. For example, GPU code is written as a single **sequential** loop body (called a *kernel*) that is run in parallel by the GPU driver.

Flat vs. Nested Data Parallelism

The second example, *smvm*, performed a **nested parallel loop**, i.e. a parallel computation inside another parallel computation. This is called ***nested data parallelism***.

Most data-parallel architectures do **not** support such things directly. For example, GPU code is written as a single **sequential** loop body (called a ***kernel***) that is run in parallel by the GPU driver.

There exists an algorithm to translate nested data-parallel programs to flat ones (***vectorisation***), but it does not always produce efficient code. In particular, it loops over the input array more often than necessary, and ***fusion*** optimisations are often required to consolidate multiple loops into one.

Another Example

Example (Parallel Quicksort)

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p      = xs[0]  
    ls     = [x | x ← xs, x < p]  
    es     = [x | x ← xs, x == p]  
    gs     = [x | x ← xs, x > p]  
    results = [qsort section | section ← [ls, gs]]  
  in results[0] ++ es ++ results[1]
```

Cost Models

A parallel algorithm gives rise two **two** time cost models:

- **Work cost** is the overall number of steps required to execute the program. Corresponds to time cost on a processor with just **one core**.
- **Depth cost** is the minimal number of **parallel** steps required to execute the program. Corresponds to the time cost on a processor with **unlimited cores**.

With these two measures, the cost for a given number of processing units can be derived.

Denotationally

Let's define two denotational cost models, $\llbracket e \rrbracket_{\mathcal{W}}$ for work cost, and $\llbracket e \rrbracket_{\mathcal{D}}$ for depth cost, and an integer for our cost measure.

Denotationally

Let's define two denotational cost models, $\llbracket e \rrbracket_{\mathcal{W}}$ for work cost, and $\llbracket e \rrbracket_{\mathcal{D}}$ for depth cost, and an integer for our cost measure. For sequential operations, the two measures coincide:

$$\begin{aligned}\llbracket \text{len } e \rrbracket_{\mathcal{W}} &= \llbracket e \rrbracket_{\mathcal{W}} + 1 \\ \llbracket \text{len } e \rrbracket_{\mathcal{D}} &= \llbracket e \rrbracket_{\mathcal{D}} + 1\end{aligned}$$

Denotationally

Let's define two denotational cost models, $\llbracket e \rrbracket_{\mathcal{W}}$ for work cost, and $\llbracket e \rrbracket_{\mathcal{D}}$ for depth cost, and an integer for our cost measure. For sequential operations, the two measures coincide:

$$\llbracket \text{len } e \rrbracket_{\mathcal{W}} = \llbracket e \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket \text{len } e \rrbracket_{\mathcal{D}} = \llbracket e \rrbracket_{\mathcal{D}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{W}}$$

Denotationally

Let's define two denotational cost models, $\llbracket e \rrbracket_{\mathcal{W}}$ for work cost, and $\llbracket e \rrbracket_{\mathcal{D}}$ for depth cost, and an integer for our cost measure. For sequential operations, the two measures coincide:

$$\llbracket \text{len } e \rrbracket_{\mathcal{W}} = \llbracket e \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket \text{len } e \rrbracket_{\mathcal{D}} = \llbracket e \rrbracket_{\mathcal{D}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{W}} = \llbracket e_1 \rrbracket_{\mathcal{W}} + \llbracket e_2 \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{D}} = \llbracket e_1 \rrbracket_{\mathcal{D}} + \llbracket e_2 \rrbracket_{\mathcal{D}} + 1$$

Denotationally

Let's define two denotational cost models, $\llbracket e \rrbracket_{\mathcal{W}}$ for work cost, and $\llbracket e \rrbracket_{\mathcal{D}}$ for depth cost, and an integer for our cost measure. For sequential operations, the two measures coincide:

$$\llbracket \text{len } e \rrbracket_{\mathcal{W}} = \llbracket e \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket \text{len } e \rrbracket_{\mathcal{D}} = \llbracket e \rrbracket_{\mathcal{D}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{W}} = \llbracket e_1 \rrbracket_{\mathcal{W}} + \llbracket e_2 \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{D}} = \llbracket e_1 \rrbracket_{\mathcal{D}} + \llbracket e_2 \rrbracket_{\mathcal{D}} + 1$$

For parallel operations, we get different results for each measure:

$$\llbracket [e_r \mid x_1 \leftarrow e_1 \mid \dots \mid x_n \leftarrow e_n] \rrbracket_{\mathcal{W}} =$$

Denotationally

Let's define two denotational cost models, $\llbracket e \rrbracket_{\mathcal{W}}$ for work cost, and $\llbracket e \rrbracket_{\mathcal{D}}$ for depth cost, and an integer for our cost measure. For sequential operations, the two measures coincide:

$$\llbracket \text{len } e \rrbracket_{\mathcal{W}} = \llbracket e \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket \text{len } e \rrbracket_{\mathcal{D}} = \llbracket e \rrbracket_{\mathcal{D}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{W}} = \llbracket e_1 \rrbracket_{\mathcal{W}} + \llbracket e_2 \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{D}} = \llbracket e_1 \rrbracket_{\mathcal{D}} + \llbracket e_2 \rrbracket_{\mathcal{D}} + 1$$

For parallel operations, we get different results for each measure:

$$\llbracket [e_r \mid x_1 \leftarrow e_1 \mid \dots \mid x_n \leftarrow e_n] \rrbracket_{\mathcal{W}} = \llbracket e_r \rrbracket_{\mathcal{W}} + \sum_i \llbracket e_i \rrbracket_{\mathcal{W}}$$

$$\llbracket [e_r \mid x_1 \leftarrow e_1 \mid \dots \mid x_n \leftarrow e_n] \rrbracket_{\mathcal{D}} =$$

Denotationally

Let's define two denotational cost models, $\llbracket e \rrbracket_{\mathcal{W}}$ for work cost, and $\llbracket e \rrbracket_{\mathcal{D}}$ for depth cost, and an integer for our cost measure. For sequential operations, the two measures coincide:

$$\llbracket \text{len } e \rrbracket_{\mathcal{W}} = \llbracket e \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket \text{len } e \rrbracket_{\mathcal{D}} = \llbracket e \rrbracket_{\mathcal{D}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{W}} = \llbracket e_1 \rrbracket_{\mathcal{W}} + \llbracket e_2 \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{D}} = \llbracket e_1 \rrbracket_{\mathcal{D}} + \llbracket e_2 \rrbracket_{\mathcal{D}} + 1$$

For parallel operations, we get different results for each measure:

$$\llbracket [e_r \mid x_1 \leftarrow e_1 \mid \dots \mid x_n \leftarrow e_n] \rrbracket_{\mathcal{W}} = \llbracket e_r \rrbracket_{\mathcal{W}} + \sum_i \llbracket e_i \rrbracket_{\mathcal{W}}$$

$$\llbracket [e_r \mid x_1 \leftarrow e_1 \mid \dots \mid x_n \leftarrow e_n] \rrbracket_{\mathcal{D}} = \llbracket e_r \rrbracket_{\mathcal{D}} + \max_i \llbracket e_i \rrbracket_{\mathcal{D}}$$

Denotationally

Let's define two denotational cost models, $\llbracket e \rrbracket_{\mathcal{W}}$ for work cost, and $\llbracket e \rrbracket_{\mathcal{D}}$ for depth cost, and an integer for our cost measure. For sequential operations, the two measures coincide:

$$\llbracket \text{len } e \rrbracket_{\mathcal{W}} = \llbracket e \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket \text{len } e \rrbracket_{\mathcal{D}} = \llbracket e \rrbracket_{\mathcal{D}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{W}} = \llbracket e_1 \rrbracket_{\mathcal{W}} + \llbracket e_2 \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{D}} = \llbracket e_1 \rrbracket_{\mathcal{D}} + \llbracket e_2 \rrbracket_{\mathcal{D}} + 1$$

For parallel operations, we get different results for each measure:

$$\llbracket [e_r \mid x_1 \leftarrow e_1 \mid \dots \mid x_n \leftarrow e_n] \rrbracket_{\mathcal{W}} = \llbracket e_r \rrbracket_{\mathcal{W}} + \sum_i \llbracket e_i \rrbracket_{\mathcal{W}}$$

$$\llbracket [e_r \mid x_1 \leftarrow e_1 \mid \dots \mid x_n \leftarrow e_n] \rrbracket_{\mathcal{D}} = \llbracket e_r \rrbracket_{\mathcal{D}} + \max_i \llbracket e_i \rrbracket_{\mathcal{D}}$$

$$\llbracket e_1 ++ e_2 \rrbracket_{\mathcal{W}}$$

Denotationally

Let's define two denotational cost models, $\llbracket e \rrbracket_{\mathcal{W}}$ for work cost, and $\llbracket e \rrbracket_{\mathcal{D}}$ for depth cost, and an integer for our cost measure. For sequential operations, the two measures coincide:

$$\llbracket \text{len } e \rrbracket_{\mathcal{W}} = \llbracket e \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket \text{len } e \rrbracket_{\mathcal{D}} = \llbracket e \rrbracket_{\mathcal{D}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{W}} = \llbracket e_1 \rrbracket_{\mathcal{W}} + \llbracket e_2 \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{D}} = \llbracket e_1 \rrbracket_{\mathcal{D}} + \llbracket e_2 \rrbracket_{\mathcal{D}} + 1$$

For parallel operations, we get different results for each measure:

$$\begin{aligned} \llbracket [e_r \mid x_1 \leftarrow e_1 \mid \dots \mid x_n \leftarrow e_n] \rrbracket_{\mathcal{W}} &= \llbracket e_r \rrbracket_{\mathcal{W}} + \sum_i \llbracket e_i \rrbracket_{\mathcal{W}} \\ \llbracket [e_r \mid x_1 \leftarrow e_1 \mid \dots \mid x_n \leftarrow e_n] \rrbracket_{\mathcal{D}} &= \llbracket e_r \rrbracket_{\mathcal{D}} + \max_i \llbracket e_i \rrbracket_{\mathcal{D}} \end{aligned}$$

$$\begin{aligned} \llbracket e_1 ++ e_2 \rrbracket_{\mathcal{W}} &= \llbracket e_1 \rrbracket_{\mathcal{W}} + \llbracket e_2 \rrbracket_{\mathcal{W}} + \text{length}(e_1) + \text{length}(e_2) \\ \llbracket e_1 ++ e_2 \rrbracket_{\mathcal{D}} & \end{aligned}$$

Denotationally

Let's define two denotational cost models, $\llbracket e \rrbracket_{\mathcal{W}}$ for work cost, and $\llbracket e \rrbracket_{\mathcal{D}}$ for depth cost, and an integer for our cost measure. For sequential operations, the two measures coincide:

$$\llbracket \text{len } e \rrbracket_{\mathcal{W}} = \llbracket e \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket \text{len } e \rrbracket_{\mathcal{D}} = \llbracket e \rrbracket_{\mathcal{D}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{W}} = \llbracket e_1 \rrbracket_{\mathcal{W}} + \llbracket e_2 \rrbracket_{\mathcal{W}} + 1$$

$$\llbracket e_1[e_2] \rrbracket_{\mathcal{D}} = \llbracket e_1 \rrbracket_{\mathcal{D}} + \llbracket e_2 \rrbracket_{\mathcal{D}} + 1$$

For parallel operations, we get different results for each measure:

$$\llbracket [e_r \mid x_1 \leftarrow e_1 \mid \dots \mid x_n \leftarrow e_n] \rrbracket_{\mathcal{W}} = \llbracket e_r \rrbracket_{\mathcal{W}} + \sum_i \llbracket e_i \rrbracket_{\mathcal{W}}$$

$$\llbracket [e_r \mid x_1 \leftarrow e_1 \mid \dots \mid x_n \leftarrow e_n] \rrbracket_{\mathcal{D}} = \llbracket e_r \rrbracket_{\mathcal{D}} + \max_i \llbracket e_i \rrbracket_{\mathcal{D}}$$

$$\llbracket e_1 ++ e_2 \rrbracket_{\mathcal{W}} = \llbracket e_1 \rrbracket_{\mathcal{W}} + \llbracket e_2 \rrbracket_{\mathcal{W}} + \text{length}(e_1) + \text{length}(e_2)$$

$$\llbracket e_1 ++ e_2 \rrbracket_{\mathcal{D}} = \llbracket e_1 \rrbracket_{\mathcal{D}} + \llbracket e_2 \rrbracket_{\mathcal{D}} + 1$$

Fold

As **fold** involves building a tree, we have a logarithmic term to consider:

$$[[\text{fold}_{\otimes} e]]_{\mathcal{W}} =$$

Fold

As `fold` involves building a tree, we have a logarithmic term to consider:

$$\begin{aligned} \llbracket \text{fold}_{\otimes} e \rrbracket_{\mathcal{W}} &= \llbracket e \rrbracket_{\mathcal{W}} + \text{length}(e) \\ \llbracket \text{fold}_{\otimes} e \rrbracket_{\mathcal{D}} &= \end{aligned}$$

Fold

As `fold` involves building a tree, we have a logarithmic term to consider:

$$\begin{aligned} \llbracket \text{fold}_{\otimes} e \rrbracket_{\mathcal{W}} &= \llbracket e \rrbracket_{\mathcal{W}} + \text{length}(e) \\ \llbracket \text{fold}_{\otimes} e \rrbracket_{\mathcal{D}} &= \llbracket e \rrbracket_{\mathcal{D}} + \log_2(\text{length}(e)) \end{aligned}$$

Quicksort Work Complexity

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p      = xs[0]  
    ls     = [x | x ← xs, x < p]  
    es     = [x | x ← xs, x == p]  
    gs     = [x | x ← xs, x > p]  
    results = [qsort section | section ← [ls, gs]  
  in results[0] ++ es ++ results[1]
```

Let $W(n)$ be $\llbracket qsort\ xs \rrbracket_W$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Work Complexity

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p      = xs[0] 1  
    ls     = [x | x ← xs, x < p]  
    es     = [x | x ← xs, x == p]  
    gs     = [x | x ← xs, x > p]  
    results = [qsort section | section ← [ls, gs]]  
  in results[0] ++ es ++ results[1]
```

Let $W(n)$ be $\llbracket qsort\ xs \rrbracket_W$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Work Complexity

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p      = xs[0] 1  
    ls     = [x | x ← xs, x < p] n  
    es     = [x | x ← xs, x == p]  
    gs     = [x | x ← xs, x > p]  
    results = [qsort section | section ← [ls, gs]]  
  in results[0] ++ es ++ results[1]
```

Let $W(n)$ be $\llbracket qsort\ xs \rrbracket_W$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Work Complexity

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p      = xs[0] 1  
    ls     = [x | x ← xs, x < p] n  
    es     = [x | x ← xs, x == p] n  
    gs     = [x | x ← xs, x > p]  
    results = [qsort section | section ← [ls, gs]]  
  in results[0] ++ es ++ results[1]
```

Let $W(n)$ be $\llbracket qsort\ xs \rrbracket_W$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Work Complexity

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p      = xs[0] 1  
    ls     = [x | x ← xs, x < p] n  
    es     = [x | x ← xs, x == p] n  
    gs     = [x | x ← xs, x > p] n  
    results = [qsort section | section ← [ls, gs]]  
  in results[0] ++ es ++ results[1]
```

Let $W(n)$ be $\llbracket qsort\ xs \rrbracket_W$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Work Complexity

```

recfun qsort :: ([Int] → [Int]) xs =
  if len xs ≤ 1 then xs
  else let
    p      = xs[0]                                1
    ls     = [x | x ← xs, x < p]              n
    es     = [x | x ← xs, x == p]             n
    gs     = [x | x ← xs, x > p]              n
    results = [qsort section | section ← [ls, gs]] 2 × W ( $\frac{n}{2}$ )
  in results[0] ++ es ++ results[1]

```

Let $W(n)$ be $\llbracket qsort\ xs \rrbracket_W$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Work Complexity

```

recfun qsort :: ([Int] → [Int]) xs =
  if len xs ≤ 1 then xs
  else let
    p      = xs[0]
    ls     = [x | x ← xs, x < p]
    es     = [x | x ← xs, x == p]
    gs     = [x | x ← xs, x > p]
    results = [qsort section | section ← [ls, gs]]
  in results[0] ++ es ++ results[1]

```

1
 n
 n
 n
 $2 \times W\left(\frac{n}{2}\right)$
 $\frac{3n}{2}$

Let $W(n)$ be $\llbracket qsort\ xs \rrbracket_W$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Work Complexity

```

recfun qsort :: ([Int] → [Int]) xs =
  if len xs ≤ 1 then xs
  else let
    p      = xs[0]
    ls     = [x | x ← xs, x < p]
    es     = [x | x ← xs, x == p]
    gs     = [x | x ← xs, x > p]
    results = [qsort section | section ← [ls, gs]]
  in results[0] ++ es ++ results[1]

```

1
 n
 n
 n
 $2 \times W\left(\frac{n}{2}\right)$
 $\frac{3n}{2}$

Let $W(n)$ be $\llbracket qsort\ xs \rrbracket_W$ for an array xs that is bisected by the pivot on each recursion (average case). Then:

$$\begin{aligned}
 W(1) &= 1 \\
 W(n) &= C_1 + C_2 n + W\left(\frac{n}{2}\right)
 \end{aligned}$$

So $W(n) \in \mathcal{O}(n \log n)$ (by the Master Theorem)

Quicksort Depth Complexity

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p      = xs[0]  
    ls     = [x | x ← xs, x < p]  
    es     = [x | x ← xs, x == p]  
    gs     = [x | x ← xs, x > p]  
    results = [qsort section | section ← [ls, gs]]  
  in results[0] ++ es ++ results[1]
```

Let $D(n)$ be $\llbracket qsort\ xs \rrbracket_D$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Depth Complexity

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p           = xs[0] 1  
    ls         = [x | x ← xs, x < p]  
    es         = [x | x ← xs, x == p]  
    gs         = [x | x ← xs, x > p]  
    results    = [qsort section | section ← [ls, gs]]  
  in results[0] ++ es ++ results[1]
```

Let $D(n)$ be $\llbracket \text{qsort } xs \rrbracket_D$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Depth Complexity

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p      = xs[0] 1  
    ls     = [x | x ← xs, x < p] 1  
    es     = [x | x ← xs, x == p]  
    gs     = [x | x ← xs, x > p]  
    results = [qsort section | section ← [ls, gs]]  
  in results[0] ++ es ++ results[1]
```

Let $D(n)$ be $\llbracket \text{qsort } xs \rrbracket_D$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Depth Complexity

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p      = xs[0] 1  
    ls     = [x | x ← xs, x < p] 1  
    es     = [x | x ← xs, x == p] 1  
    gs     = [x | x ← xs, x > p]  
    results = [qsort section | section ← [ls, gs]]  
  in results[0] ++ es ++ results[1]
```

Let $D(n)$ be $\llbracket \text{qsort } xs \rrbracket_D$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Depth Complexity

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p      = xs[0]           1  
    ls     = [x | x ← xs, x < p] 1  
    es     = [x | x ← xs, x == p] 1  
    gs     = [x | x ← xs, x > p] 1  
    results = [qsort section | section ← [ls, gs]]  
  in results[0] ++ es ++ results[1]
```

Let $D(n)$ be $\llbracket \text{qsort } xs \rrbracket_D$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Depth Complexity

```
recfun qsort :: ([Int] → [Int]) xs =  
  if len xs ≤ 1 then xs  
  else let  
    p      = xs[0]                1  
    ls     = [x | x ← xs, x < p]  1  
    es     = [x | x ← xs, x == p]  1  
    gs     = [x | x ← xs, x > p]  1  
    results = [qsort section | section ← [ls, gs]]  D( $\frac{n}{2}$ )  
  in results[0] ++ es ++ results[1]
```

Let $D(n)$ be $\llbracket \text{qsort } xs \rrbracket_D$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Depth Complexity

```

recfun qsort :: ([Int] → [Int]) xs =
  if len xs ≤ 1 then xs
  else let
    p      = xs[0]                                1
    ls     = [x | x ← xs, x < p]                  1
    es     = [x | x ← xs, x == p]                1
    gs     = [x | x ← xs, x > p]                  1
    results = [qsort section | section ← [ls, gs]]   $D\left(\frac{n}{2}\right)$ 
  in results[0] ++ es ++ results[1]              2

```

Let $D(n)$ be $\llbracket qsort\ xs \rrbracket_D$ for an array xs that is bisected by the pivot on each recursion (average case).

Quicksort Depth Complexity

```

recfun qsort :: ([Int] → [Int]) xs =
  if len xs ≤ 1 then xs
  else let
    p      = xs[0]                                1
    ls     = [x | x ← xs, x < p]              1
    es     = [x | x ← xs, x == p]            1
    gs     = [x | x ← xs, x > p]              1
    results = [qsort section | section ← [ls, gs]]   $D(\frac{n}{2})$ 
  in results[0] ++ es ++ results[1]             2
  
```

Let $D(n)$ be $\llbracket qsort\ xs \rrbracket_D$ for an array xs that is bisected by the pivot on each recursion (average case). Then:

$$\begin{aligned}
 D(1) &= 1 \\
 D(n) &= C + D\left(\frac{n}{2}\right)
 \end{aligned}$$

So $D(n) \in \mathcal{O}(\log n)$ (by the Master Theorem)