Introduction

1 Imperative versus Functional Programming

1.1 Functions: Imperative versus Pure

By now, you have probably heard the phrases "functions are pointers" and "functions are values" floating around. In (pure) functional programming, functions are functions—that is, code functions, defined in your functional programming language of choice, (usually) correspond to mathematical functions. For example, the Fibonacci function could be defined as an infinite set of ordered pairs

$$\{(0,1),(1,1),(2,2),(3,3),(4,5),(5,8)\dots\}$$

It also could be defined in pseudocode as follows:

```
fun f(n : int) -> int:
  if (n <= 1): return 1
  else: return f(n - 1) + f(n - 2)
```

In pure-functional programming, this counts as a function, since every natural number input maps to one output and the code does not have any other side effects[1]. On the other hand, consider the following *imperative function*.

```
x : int = 0
y : int = 0

fun f(n : int) -> int:
  x ← 23;
  if (n <= 1): return 1;
  else: return f(n - 1) + f(n - 2) + y * 3;
```

This isn't a mathematical function for a couple of reasons. First, it affects the environment by setting the value of x on its first line. The output of this function for a particular input also isn't consistent, since the value of y might change as a program executes. This especially poses a problem for parallel algorithms, which might access or mutate some variable simultaneously.

Definition: Pure functions

A **pure function** is one that always returns the same output given the same input and does not have any side effects.

[1]This assumes an idealized model of computation in which execution always terminates and resource limits (e.g., stack overflow) and undefined behavior (e.g., integer overflow) do not occur!

In general, as long as they terminate and do not raise exceptions, pure functions without side effects are mathematical functions. The advantage of pure functions is that they can run independently without interfering with each other. If we have pure functions f and g, then a call to $f(a)$ and a call to $g(b)$ can be run in parallel! However, if these two functions are allowed to modify the shared state, then the results of the functions might depend on which one runs first. This is called **nondeterminism**. Pure functions also have the benefit of making it easier for us to reason about our code—something that is not always afforded by other parallel programs.

Even if we need state, it is still possible to write code that we can reason about:

```
fun fact(n : int) -> int:
  r : int = 1
  for i in 1...n:
    r ← r*i
  return r
```

Even though the loop contains a side effect where the value of r is modified, this function is pure. No changes to external state can affect the execution of this function, and mutation of the function's own local variables cannot escape the function's body and cause external side effects. These kinds of "internal" side effects are sometimes called "benign" side effects.

In 15-210, we will strive to write pure functions as much as possible, since pure code is easier to reason about. We will however occasionally write impure, imperative-style code where it is more appropriate for the problem at hand, particularly for *sequential algorithms*, since our focus on purity is most important when reasoning about parallelism.

1.2 Imperative versus Functional Data Structures

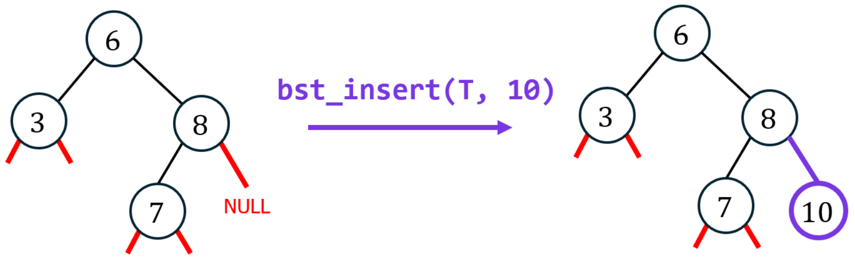
In your previous classes, you have studied data structures, and indeed the *same* data structures in two different contexts: imperative and functional. Recall the *binary search tree* (BST) data structure, which we will study even more in this class.

Imperative data structure updates

You might recall the following implementation (written in C) of insertion into a binary search tree in an imperative program.

```
// source: 15-122
tree* bst_insert(tree* T, entry e) {
  if (T == NULL) return leaf(e);
  int cmp = key_compare(entry_key(e), entry_key(T->data));
  if (cmp == 0) T->data = e;
  else if (cmp < 0) T->left = bst_insert(T->left, e);
  else T->right = bst_insert(T->right, e);
  return T;
}
```

What makes this code *imperative* is that it **modifies** the given data structure. Consider the picture below, starting with a BST containing the keys $\{3,6,7,8\}$. The operation `insert(T, 10)` creates a new leaf node containing the key 10 and then mutates the right-child pointer of the 8 node, which was previously `NULL`, to the new node.



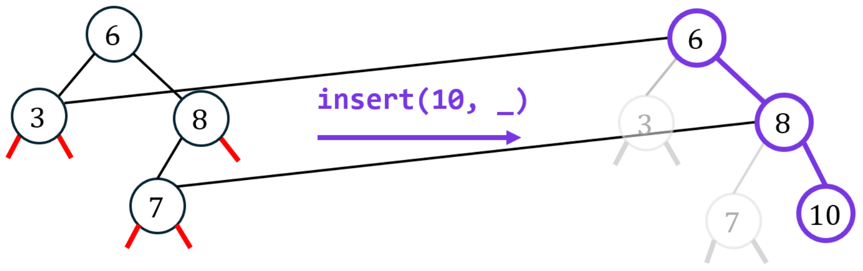
Functional data structure updates

You may also recall the following different implementation (written in Standard ML) of insertion into a binary search tree in a purely functional program.

```
(* source: 15-150 *)
fun insert (k, v) Empty =
    Node (Empty, (k, v), Empty)
  | insert (k, v) (Node (L, (k', v'), R)) =
      case Key.compare (k, k') of
        EQUAL   => Node (L, (k, v), R)
      | LESS    => Node (insert (k, v) L, (k', v'), R)
      | GREATER => Node (L, (k', v'), insert (k, v) R)
```

In contrast to the imperative implementation above, we notice one big difference. No modifications to the existing BST take place. Instead, **new nodes** are created along the insertion path. Consider again running `insert(10, _)`.

As was also the case in the imperative code, a new node must be created for 10 of course, but since we can not mutate the right child of the 8 node, the code is forced to create a new 8 node that has 10 as its right child. Similarly, since the right child of the 6 node can not be mutated to the new 8 node, it must create a new 6 node with the new 8 node as its right child as well.



If this resulted in making a copy of the entire BST, this would be very inefficient, and we would probably avoid using this style of programming. However, what makes this algorithm efficient is that it only creates new nodes for those on the insertion path. For all of the subtrees that are not modified, the new tree can simply **share** the old ones as depicted in the picture above.

1.3 Persistent Data Structures

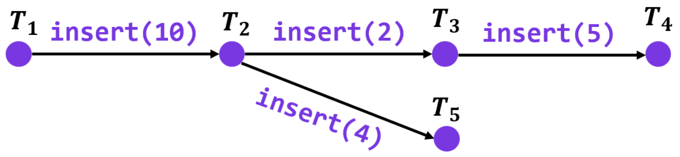
This sharing property of functional data structures not only makes them efficient, but also gives rise to an incredibly useful property, which is that they are naturally **persistent**.

Definition: Persistent Data Structure

A persistent data structure is a data structure that preserves the old versions of itself when it is updated.

That is, after we perform `insert(10, _)` and obtain the new BST containing $\{3, 6, 7, 8, 10\}$, the old BST which only contained $\{3, 6, 7, 8\}$ **still exists** and can continue to be queried, or even updated to produce another BST! This turns out to be extremely useful for implementing several important algorithms, one of which we will see later in the course.

Another way to think of this is that persistence is like having version control (e.g., git) for your data structure! You can go back and look at old versions, and even update them to branch off into separate non-linear histories.



Note that you can also implement persistent data structures in an imperative way (and indeed, you can sometimes make them even more efficient by doing so), but it takes substantial additional effort in the algorithm design and programming. The amazing thing about functional data structures is that they come persistent out of the box!

2 Abstraction: Interfaces and ADTs

In this course, we study algorithms at a level of abstraction that separates *what* an algorithm does from *how* it is implemented. This separation is essential for reasoning about correctness, efficiency, and parallelism in a clean and modular way.

2.1 Interfaces and Abstract Data Types

Definition: Interface

An **interface** specifies a collection of operations together with their intended behavior. It describes what operations are available and what they are expected to do, but deliberately does not describe how those operations are implemented.

Definition: Abstract Data Type

An **abstract data type** (ADT) is a particular kind of interface that describes a data structure. An ADT specifies the operations that can be performed on the data structure and the meaning of those operations, while leaving the underlying representation unspecified. In this sense, abstract data types are a special case of interfaces.

An *implementation* provides concrete code for the operations specified by an interface. A *data structure* is an implementation of an abstract data type, consisting of both a representation of state together with algorithms that implement the ADT's operations. More generally, algorithms themselves can be viewed as implementations of interface operations.

Throughout this course, we will use the term *interface* in this broad sense. Some interfaces describe data structures, while others describe collections of operations or algorithmic building blocks. In all cases, the key idea is the same: algorithms are written against interfaces, not implementations.

Remark: Why Interfaces Matter for Algorithms

Writing algorithms against interfaces allows us to reason about them independently of low-level implementation details. An algorithm depends only on the behavior promised by the interface it uses, not on how that behavior is realized internally.

This separation supports several important goals. First, it enables *modularity*: implementations can be improved or replaced as long as they continue to satisfy the same interface. Second, it simplifies *reasoning*: correctness arguments and cost analyses can be carried out assuming only the interface guarantees. Finally, it promotes *reuse*: the same algorithm can be applied to multiple implementations that satisfy the interface.

These benefits are especially important in parallel algorithms, where we want to express parallelism and analyze cost without reasoning about unnecessary low-level details.

2.2 The Sequence ADT

The central abstraction used throughout this course is the *sequence abstract data type*. Nearly all of the parallel algorithms we study are expressed in terms of sequences and operations on them. Rather than reasoning about arrays, pointers, or memory layouts directly, we design algorithms against the sequence interface.

Definition: Sequence

A **sequence** of length n over elements of type T is an ordered collection of values that can be viewed as a mapping from the indices $\{0, 1, \ldots, n-1\} \rightarrow T$.

Intuitively, a sequence represents an ordered collection of elements that can be accessed by position. The definition above captures the essential properties of sequences that algorithms rely on: a well-defined length and a notion of indexing. It does not prescribe how the sequence is represented in memory. The sequence ADT is then defined as

Interface: The Sequence ADT

A sequence represents a finite, ordered collection of elements that supports efficient (i.e., constant-time) random access. Conceptually, a sequence behaves like an array or a contiguous view into an array.

Sequence Access:

- `nth : (S : sequence<T>, i : int) -> T`
  Return the i^{th} element (zero-indexed) of the sequence S.
- `length : (S : sequence<T>) -> int`
  Return the length (number of elements in) the sequence S.

The `nth` function is usually abbreviated as `S[i]`, and `length` is abbreviated as `|S|`.

Subsequence Views:

- `subseq : (S : sequence<T>, i : int, k : int) -> sequence<T>`
  Returns a *view* of the subsequence of `S` starting at index i with length k.

The function `subseq` does not copy elements, otherwise it would be inefficient.

When we write functions that take a sequence, we only rely on the interface. Such functions are generic: they work for any implementation that supports random access and subseq.

split_mid We will implement many *divide-and-conquer* algorithms which will need to take a sequence and split it into two equal sized halves. For this reason, we define a helper function `split_mid(S)` which returns `(subseq(S, 0,` $\lfloor |S|/2 \rfloor$`), subseq(S,` $\lfloor |S|/2 \rfloor$`,` $\lceil |S|/2 \rceil$`))`

2.2.1 A concrete sequence: ArraySequence

We now fix a concrete sequence representation that we will use whenever an algorithm needs to *create* a sequence. For all of the algorithms we design in this class, we will assume that any sequences we explicitly construct are `ArraySequence<T>` values, a sequence backed by a contiguous array of elements of type `T` which also stores its own length. We will assume that this is the type returned by our pseudocode sequence comprehension. We also define the function `tabulate`, where `tabulate(f, n)` returns the sequence `[f(x) for x in 0...n-1]`, i.e., the sequence containing $f(0), f(1), \ldots, f(n-1)$ in that order.

- `tabulate : ((int -> T), int) -> ArraySequence<T>`

For efficiency, the `subseq` function would not make a new array and copy the elements of the subsequence. Instead, it would simply store a reference to the underlying `ArraySequence<T>` as well as a “left” and “right” position, indicating the boundaries of the slice. With this concrete implementation, all operations, `nth`, `length`, `subseq` take $O(1)$ time.

We will also use Python-like list-comprehension syntax in our pseudocode since it is nice and recognizable, so in place of `tabulate(f, n)` we may write

```
parallel [f(i) for i in 0...n-1]
```

Remark: Mutation and the Sequence ADT

The sequence ADT, as presented here, is purely functional: none of the operations mutate the sequences. This design choice is intentional. Pure, functional interfaces are significantly easier to reason about, especially in the presence of parallelism, since they avoid interference between concurrent computations.

That said, mutation is not forbidden. In this course, we freely use mutable data structures and in-place updates when designing *sequential* algorithms, where such reasoning is straightforward. However, when designing *parallel* algorithms, we will almost always work with functional sequence operations, and only resort to mutation in rare cases where it is truly necessary.

This discipline reflects a broader principle: mutation is a powerful tool, but one that must be used carefully. By default, we favor functional abstractions that make correctness and parallel cost reasoning clean and modular.

3 Abstraction: Models of Computation and Cost Models

3.1 Sequential Cost Model: Word RAM

Before discussing parallel algorithms, it is helpful to recall the cost model we implicitly use when analyzing sequential algorithms. Throughout most of computer science, this baseline model is the **word RAM** model. Most algorithm analysis you did in previous classes was probably in the word RAM model, even if you didn't call it that explicitly.

Definition: Word RAM

In the word RAM (Random Access Machine) model, computation proceeds sequentially, and the machine operates on fixed-size *machine words* which are w-bits long.

- We assume that the machine word size is large enough to store any input value and any index into the input (in particular, $w = \Omega(\log n)$ for inputs of size n).
- Each basic operation on w-bit values, such as arithmetic operations, comparisons, branching, and reading or writing memory, costs $O(1)$.
- Memory is assumed to be randomly accessible, meaning that accessing any memory location takes constant time.

This model closely reflects the behavior of real machines at a high level, while remaining simple enough to support asymptotic analysis. When we say that a sequential algorithm runs in $O(n)$ time, we typically mean that it performs $O(n)$ word-RAM operations.

The word RAM model provides a clear and stable notion of cost for sequential computation. However, it does not directly extend to parallel algorithms, where multiple computations may proceed simultaneously. In the following sections, we develop abstractions and cost measures that allow us to reason about parallel computation in a similarly machine-independent way.

Remark: Why word RAM?

The word RAM much better models the capabilities of real computers, which naively perform fixed-precision arithmetic. For example, trying to compute large numbers in languages like C, Java, C++, etc., will quickly lead to *overflow*, and doing so in Python will lead to large numbers which are neither constant time not space to operate on.

A naïve unit-cost RAM model that allows arithmetic on arbitrarily large integers (i.e. unlimited precision arithmetic) is also unrealistically powerful: such a model can perform computations that are believed to be intractable, for example solving NP-hard problems in polynomial time. Restricting computation to fixed-size machine words avoids this issue and better reflects the capabilities of real computers.

3.2 Nested Parallel Programs

Our model of parallelism will be an abstraction at a higher level than an actual parallel computer, hiding details such as cores and scheduling. Instead, we model parallelism with **nested fork-join parallelism**. We extend the word RAM model with the ability to **fork**: where the current computation splits into multiple child computations that are eligible to be run in parallel.

Definition: Fork–Join RAM

The **Fork–Join RAM** is a model extending the word RAM with structured parallelism. Computation proceeds via constant-time word-RAM operations and *fork* operations.

A fork operation creates a fixed number of child computations that may execute in parallel. The parent computation suspends at the fork point and resumes only after all of its child computations have completed; this resumption point is called the *join*. Child computations may themselves perform fork operations, allowing parallelism to be nested.

Remark: Nested parallelism as an abstraction

Early work on parallel algorithms used the *PRAM (Parallel RAM) model*, which requires algorithms to be written for a fixed number of processors. This forces the programmer to explicitly decide which computations execute on which processor at which times (a task known as *scheduling*). While powerful, this makes many otherwise simple algorithms unnecessarily complicated. Nested parallelism provides a higher-level abstraction that allows algorithms to express parallel structure without specifying how computations are assigned to processors; this complexity is instead handled by the implementation.

Abstracting away these kinds of details about how algorithms are mapped to actual hardware is not unusual, in fact, it is central to how we program! For example, when writing code we use variables rather than manually managing memory addresses. We write `int x = 5` without specifying where in memory the value is stored; the compiler and operating system handle those details for us. Nested parallelism plays an analogous role for parallel algorithms: it lets us describe *what* can run in parallel, while leaving *how* it is carried out on the actual processor cores to the underlying system.

3.2.1 Parallelism in our pseudocode

In our pseudocode, we express fork–join parallelism using the **`parallel`** keyword. Conceptually, a **`parallel`** expression corresponds to a fork in the Fork–Join RAM model: the current computation forks into child computations to evaluate the elements of the expression in parallel, and automatically joins once all have completed. Syntactically, the **`parallel`** keyword can be applied to *tuples* or *sequence* comprehensions.

Parallel tuple evaluation We can evaluate a tuple of expressions in parallel with the parallel keyword. For example, to compute the functions $f(a)$ and $g(b)$ in parallel, we could write

```
x, y = parallel (f(a), g(b))
```

Parallel sequence comprehension This construct evaluates $f(x)$ for all $x \in A$ in parallel and returns their results as a `sequence` in the same order once all have completed.

```
parallel [f(x) for x in A]
```

In other words, this is equivalent to the higher-order function `map(f, A)` where `f : U → V` is a pure function and A is a `sequence<U>`, using syntax resembling Python's list comprehension.

Remark: Parallel loops

In practice, parallel programs, especially in systems languages such as C++, often use constructs like parallel loops that execute side-effecting code across many iterations concurrently. These are extremely powerful and widely used in production systems. However, reasoning about the correctness of such code requires careful attention to shared state, interference, and synchronization.

In this course, we will largely avoid explicit side-effecting parallel loops. Instead, we focus on structured parallelism and higher-level abstractions that allow algorithms to be expressed and reasoned about cleanly and deterministically. Imperative parallel loops may be used internally by library implementations, but they will not be a primary tool for expressing algorithms in this class.

3.3 A parallel cost model: Work and Span

In the sequential setting, the word RAM model gives us a clear notion of cost: an algorithm's running time is the number of constant-time operations it performs. Parallel algorithms require a refinement of this idea, since multiple computations may proceed simultaneously. Measuring the running time of a parallel program would seem to require knowledge of the number of processors P it runs on, making the running time a function of P. Doing so would again require thinking about machine-specific details and scheduling, which we would prefer to avoid. An elegant solution to this is to measure not just one quantity, but two. We describe the cost of a parallel computation using two quantities: its **work** and its **span**.

Intuitively, *work* measures the total amount of computation performed, while *span* measures the length of the longest chain of dependencies that must be executed sequentially. These two quantities capture complementary aspects of parallel performance.

3.3.1 Work

Definition: Work

The work of a computation is the total number of constant-time operations it performs, counting all operations across all parallel branches. In other words, work measures how much computation would be performed if all parallelism were executed sequentially.

For example, when two computations are executed in parallel, the total work is the sum of the work of the two computations. Sequential composition likewise adds work. Work therefore generalizes the familiar notion of running time from the word RAM model to parallel programs.

3.3.2 Span

Definition: Span

The span of a parallel computation measures its inherent sequentiality. It is defined as the length of the longest chain of computations that must occur one after another due to dependencies. In other words, span measures the running time of the algorithm if it had infinitely many processors with no scheduling overhead.

When two computations are executed sequentially, their spans add. When they are executed in parallel, their spans combine by taking the maximum, since the two computations may proceed concurrently. The span thus captures the critical path through the computation.

This structural approach allows us to analyze parallel algorithms compositionally, using simple recurrences that closely mirror the algorithm itself.

3.4 Example: Parallel Sum

Consider the following simple implementation of a parallel algorithm that computes the sum of an integer sequence (i.e., in fancy words, a *reduction* using the plus operation).

Algorithm: Parallel Sequence Sum

```
fun sum(S : sequence<int>) -> int:
  match length(S) with:
    case 0: return 0        // Empty sequence
    case 1: return S[0]     // Singleton sequence
    case _:
      L, R = split_mid(S)   // Helper function
      Lsum, Rsum = parallel (sum(L), sum(R))
      return Lsum + Rsum
```

Work analysis We can model the work of this algorithm using a recurrence relation. Let $W_{\text{sum}}(n)$ be the work required to evaluate `sum(S)` for a sequence of length n. The function performs constant work in the base cases (when $n \le 1$). Otherwise, it performs constant work to do `split_mid` and to sum the results, plus the work of two recursive calls, each of size $n/2$ (ignoring rounding errors). Thus we can write the recurrence

$$W_{\text{sum}}(n) = \begin{cases} O(1) & \text{if } n \leq 1, \\ 2 \cdot W_{\text{sum}}(n/2) + O(1) & \text{otherwise.} \end{cases}$$

We will review solving recurrence relations in depth next lecture, but for now, you might recall from previous classes that we can solve this recurrence by unrolling it to get

$$W_{\text{sum}}(n) = c\left(1 + 2 + 4 + \ldots + \frac{n}{2} + n\right) = O(n).$$

Span analysis We can model the span similarly. Let $S_{\text{sum}}(n)$ denote the span of `sum(S)` for a sequence of length n. The algorithm performs constant work plus a parallel call to two recursive problems of size $n/2$. The span of the algorithm is the *maximum* of the two parallel recursive calls, which have span $S(n/2)$, but since they are of the same size, its just $S(n/2)$. Therefore, we get a recurrence relation like

$$S_{\text{sum}}(n) = \begin{cases} O(1) & \text{if } n \leq 1, \\ W_{\text{sum}}(n/2) + O(1) & \text{otherwise.} \end{cases}$$

Unrolling this recurrence relation, we get a solution of $S_{\text{sum}}(n) = O(\log n)$.

Remark: Nested Fork–Join and Work–Span are the right abstraction

Work and span together provide a machine-independent description of parallel cost. Work captures total effort, while span captures available parallelism. Any parallel execution must take time at least proportional to the span, regardless of the number of processors available.

Suppose we run a computation with work W and span S on a machine with P processors. Even with perfect load balancing, the computation must take at least $\Omega(W/P)$ time, since a total of W work must be performed. Moreover, the span S is by definition a lower bound on the running time, since the longest chain of dependent computations must execute sequentially. Thus, any execution must take time at least

$$\Omega\left(\max\left(\frac{W}{P}, S\right)\right).$$

A fundamental result in parallel algorithms, **Brent's Theorem**, shows that this lower bound is essentially achievable: a computation with work W and span S can be executed on a P-processor machine in time

$$O\left(\max\left(\frac{W}{P}, S\right)\right).$$

We will prove this theorem later in the course. For now, it suffices to note that this result vindicates our use of nested fork–join parallelism and work–span analysis: algorithms designed at this level of abstraction can be realized on concrete machines with asymptotically optimal performance!