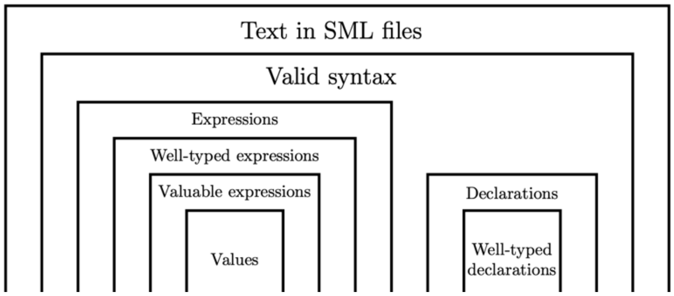

	Recurrence	Asymptotic Upper Bound
$T(n) =$	$T(n-1)+c$	$O(n)$
$T(n) =$	$T(n/2)+c$	$O(\log n)$
$T(n) =$	$2T(n/2)+c$	$O(n)$
$T(n) =$	$T(n/2)+c_1 n + c_0$	$O(n)$
$T(n) =$	$2T(n/2)+c_1 n + c_0$	$O(n \log n)$
$T(n) =$	$T(n-1)+c_1 n + c_0$	$O(n^2)$
$T(n) =$	$2T(n-1)+c$	$O(2^n)$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \qquad \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{\log n} \frac{1}{2^i} \le \sum_{i=0}^{\infty} \frac{1}{2^i} = 2 \in O(1) \qquad \sum_{i=0}^{n-1} a^i = \frac{1-a^n}{1-a}$$

function declaration binds identifier f to **closure**: lambda expression + environment (all bindings when f declared)

```
val x: int = 1
fun f (y : int):int = x + y   [1\x]
fun g (x : int):int = x + 2   [1\x, fn y => 1 + y \f]
val x: int = 2                [(fn x => x + 2) \g]
val ans: int = (f x) + g x    [2\x]
f x = 3, g x = 4, ans = 7
```

- if $e_1 \hookrightarrow v$ and $e_2 \hookrightarrow v$ with v a value, then $e_1 \cong e_2$
- if $e_1 \Longrightarrow e_2$, then $e_1 \cong e_2$
- if $e_1 \Longrightarrow e$ and $e_2 \Longrightarrow e$ with e an expression, then $e_1 \cong e_2$
- a well-typed expression e is **valuable** if there exists a value v such that $e \Longrightarrow v$ (e evaluates to a value)
- a WT expression `e1 : t1 → t2`, for types `t1` and `t2`, is **total** if for all values `e2 : t1`, expression `e1 e2` is valuable
- a function is **tail recursive** if it is recursive and performs no computations after calling itself recursively (tail calls): more *space* consistent, but not necessarily more *time* efficient
  - `fun length ([] : int list) : int = 0`
    `| length (x :: xs) = 1 + length xs`
  - `fun h ([] : int list, acc : int) = acc`
    `| h (x :: xs, acc : int) = h (xs, acc + 1)`
    `fun length (L : int list) = h (L, 0)`
- convert to infix: **`infixr @`**
- infix operator @ (append) is right-associative
- infix operator `::` (cons) is left-associative
- datatype (keyword) <type> = `constructor` that can take an argument of a type (*constructors aren't functions*)
  - finite constructors, but can have infinite values
- `type point = int * int` for existing types (new declaration shadows previous, for abstraction), **1 constructor**

op converts binary infix operator (*, +) to binary prefix operation

- `val tuple as (a,b) : int * int = (1,2)`

as (between variable and structured pattern) to reference a structured value both as a whole and by its constituents

- $f(n)$ is $O(g(n))$: $\exists N, c$ such that for all $n \ge N$, $f(n) \le c \cdot g(n)$
- $\log_b b^x = x$, $\log_a m = \log_b m / \log_a b$

- purpose of citing totality: justify valuability of expression containing the application of the function

Work: worst-case # of steps it takes to evaluate code sequentially

- X @ Y has O(n) work, with n the length of X
- recursive calls cased on: include in analysis

Span: … evaluate code in parallel, given infinite processors

Input size: unit by which we measure the size of a value, which is used to quantify work/span analysis ($d = \log n \Leftrightarrow n = 2^d$)

- $O(\log n) \subset O(n) \subset O(n^2) \subset \cdots$

	list sort	list merge sort	tree merge sort
work	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
span	$O(n^2)$	$O(n)$	$O((\log n)^{3\text{ (or 2)}})$

- **Base Case**: non-recursive constructors of the datatype (`[]`)
- **Inductive Case**: recursive constructors (`::`)
  - **Inductive Step**: `x :: xs`
  - **Induction Hypothesis**: theorem holds for xs
- **Simple**: if $P(k)$, then $P(k+1)$
  - **IS**: fix $n = k+2$ for some $k \in \mathbb{N}$
  - **IH**: assume $P(k)$, want to show $P(n)$
- **Strong**: if $P(k)$ for all k, $0 \le k < n$, then $P(n)$

```
fun power (n : int, 0 : int) : int = 1
  | power (n, k) = n * power (n, k-1)
```

Proving correctness – Theorem: for all values n : int and k : int, with k ≥ 0, power(n, k) $\hookrightarrow n^k$

Proof: By standard induction on k

Base case: k = 0

Need to show: power(n, 0) $\hookrightarrow n^0$ for all values n : int

Showing: power(n, 0) = 1

Inductive case: step from k to k+1, **for some value** k : int, k ≥ 0.

Induction hypothesis: power(n, k) $\hookrightarrow n^k$ for some values n : int

Need to show: power(n, k+1) $\hookrightarrow n^{k+1}$ for all values n : int

Showing: power(n, k+1)

$\Longrightarrow$ n * power(n, k+1-1) [2nd clause power, k+1> 0]

$\Longrightarrow$ n * n^k [IH]

The base case and induction step establish the Theorem by the principle of Mathematical Induction.

```
fun flatten ([] : int list list) : int list = []
  | flatten (L :: LS) = (case L of
      [] => flatten LS
    | x :: xs => x :: (flatten (xs :: LS)))
```

We want to show that for all values `LL : int list list`, `flatten LL ≅ oldFlat LL`.

Proof: We proceed by structural induction on `LL`

Base Case: `LL = []` … Hence, `flatten [] ≅ oldFlat []`

Inductive Case: `LL = L :: LS` for some values `L : int list` and `LS : int list list`

Inductive Hypothesis: Assume `flatten LS ≅ oldFlat LS`

Want to show: `flatten L :: LS ≅ oldFlat L :: LS`

We then proceed by structural induction on `L`

Inner Base Case: Let `L ≅ []` …

Inner Inductive Case: Let L ≅ `x::xs` for some x : int and xs : int list

Inner IH: `flatten (xs :: LS) ≅ oldFlat (xs :: LS)`

WTS: `flatten((x::xs)::LS) ≅ oldFlat ((x::xs)::LS)`

[cite outer/inner IH, **totality**, clause # of function]

Since `flatten L :: LS ≅ oldFlat L :: LS`, by structural induction on LL, for all LL : int list list, flatten LL ≅ oldFlat LL.

```
fun leaves Nub = []
  | leaves (Branch (L, C, R, v)) =
        (case (leaves L, leaves C, leaves R) of
               ([] ,[] ,[]) => [v]
              | (resL, resC, resR) =>
                   (resL @ resC) @ resR)
fun accleaves (Nub : tritree, L : int list) = L
  | accleaves (Branch(Nub, Nub, Nub, v), L) = v :: L
  | accleaves (Branch(l, c, r, v), L) =
        accleaves(l, accleaves(c, accleaves(r, L))
```

Let n be the number of Branches in the tree.

- $W_{\text{leaves}}(0) = k_0,\ W_{\text{leaves}}(1) = k_1$
- $W_{\text{leaves}}(n) = W_{\text{leaves}}(n_l) + W_{\text{leaves}}(n_c) + W_{\text{leaves}}(n_r) + W_{@}(n_l) + W_{@}(n_l + n_c) + k_2 = 3W_{\text{leaves}}\left(\frac{n}{3}\right) + kn$

There are $\log_3 n$ levels, the work/node at level i is $\frac{kn}{3^i}$.

There are 3^i nodes at level i.

$$\sum_{i=0}^{\log_3 n} 3^i \cdot \frac{kn}{3^i} = kn \log_3 n \Longrightarrow W_{\text{leaves}}(n) \in O(n \log n)$$

- $S_{\text{leaves}}(n) = \max\left(S_{\text{leaves}}(n_l), S_{\text{leaves}}(n_c), S_{\text{leaves}}(n_r)\right) + \max\left(S_{@}(n_l), S_{@}(n_l + n_c)\right) + k_2 = S_{\text{leaves}}\left(\frac{n}{3}\right) + kn$

There are $\log_3 n$ levels, the work/node at level i is $\frac{kn}{3^i}$.

There is 1 node at level i.

$$\sum_{i=0}^{\log_3 n} \frac{kn}{3^i} \le \sum_{i=0}^{\infty} \frac{kn}{3^i} = \frac{kn}{1 - \frac{1}{3}} = \frac{3kn}{2} \Longrightarrow S_{\text{leaves}}(n) \in O(n)$$

- $W_{\text{accleaves}}(n) = W_{\text{accleaves}}(n_l) + W_{\text{accleaves}}(n_c) + W_{\text{accleaves}}(n_r) + k_2 = 3W_{\text{accleaves}}\left(\frac{n}{3}\right) + k_2$

There are $\log_3 n$ levels, the work/node at level i is $\frac{kn}{3^i}$.

There are 3^i nodes at level i.

$$\sum_{i=0}^{\log_3 n} 3^i k_2 = k_2 \frac{3n-1}{2} \Longrightarrow W_{\text{accleaves}}(n) \in O(n)$$

We conduct the analysis only in terms of the number of Branches in the tree, and not the length of the list L in the 2nd argument, since at no point does the function depend on the value of L, so the length of L does not affect the work or span of the function.

- $S_{\text{accleaves}}(n) = S_{\text{accleaves}}(n_l) + S_{\text{accleaves}}(n_c) + S_{\text{accleaves}}(n_r) + k_2 = 3S_{\text{accleaves}}\left(\frac{n}{3}\right) + k_2$

The analysis is identical to the work because the recursive calls to the function are sequential, so $S_{\text{accleaves}}(n) \in O(n)$.

function binding vs lambda expressions:

```
val add : int * int -> int = fn (x,y) => x + y
fun add (x : int,y : int) : int = x + y
```

- *clausal <u>patterns</u>* (true, GREATER, _) must be able to match to the type of the expression being cased on
- *clausal <u>expressions</u>* ((), ":)", true) must all have the same type
- a pattern that accounts for every possible value of the type it matches to (bool, order) performs an exhaustive match
- Non-value expressions (2 + 2) are not valid patterns

```
val () = Test.string_int("test_1", ("two", 2), f 2)
val () = Test.int_list_eq("test_2", [1, 3], g 2)
```

- `div`, `mod`: integer division, mod, `/`: real division
- `[]`: both int list and int list list
- "by *<u>type inference</u>*, the first argument to v should have type int, but not x has type bool; no value since ill-typed"

```
fun helper ([]:bool list, _:bool list, Ls:bool list list):
        bool list list = Ls
  | helper (true :: xs, seen, Ls) = helper (xs,
        seen @ [true], (seen @ (false :: xs)) :: Ls)
  | helper (false :: xs, seen, Ls) = helper (xs,
        seen @ [false], Ls)
(* flipOne L ==> An ordered list containing all the ways to
flip exactly one true to false (bool list list) *)
fun flipOne (L : bool list) = helper (L, [], [])
```

```
fun sum [] = 0
  | sum (x :: xs) = x + sum xs
(* sublistSum(L,n) ==>* SOME(L') where L' is a sublist of L
which sums to n, or NONE if no such sublist *)
fun sublistSum (L : int list, 0 : int) = SOME []
  | sublistSum ([], n) : int list option = NONE
  | sublistSum (x :: xs : int list, n : int) =
        if ((x + sum xs) = n) then SOME (x :: xs)
        else (case sublistSum(xs, n - x) of
                NONE => sublistSum(xs, n)
              | SOME xss => SOME (x :: xss))
```

```
(* ins (x, L) ==> a sorted permutation of x :: L *)
fun ins (x, []) = [x]
  | ins (x, y :: ys) = (case compare(x, y) of
        GREATER => y :: ins(x, ys)
        | _     => x :: y :: ys)
fun isort [] = []
  | isort (x :: xs) = ins (x, isort xs)
```

$$W_{ins}(n) = c_1 + W_{ins}(n-1) \mid c_2 \Longrightarrow W_{ins}(n) \in O(n)$$

$$W(n) = c_1 + W(n-1) + W_{\text{ins}}(n-1) \le c_1 + c_2 n + W(n-1) \Longrightarrow O(n^2)$$

```
(* msort L returns a sorted permutation of L *)
fun msort ([] : int list) : int list = []
  | msort [x] = [x]
  | msort L =
     let val (A, B) : int list * int list = split L
     in merge (msort A, msort B) end
(* split L ≅ (A, B) with A @ B a permutation of L and with
   |length A - length B| <= 1 *)
fun split ([]: int list) : int list * int list = ([], [])
  | split [x] = ([x], [])
  | split (x :: y :: rest) =
     let val (A, B) : int list * int list = split rest
     in (x :: A, y :: B) end
(* merge (A, B) returns a sorted permutation of A @ B *)
fun merge ([] : int list, B : int list) : int list = B
  | merge (A, []) = A
  | merge (x :: A, y :: B) = (case Int.compare(x, y) of
     LESS => x :: merge (A, y :: B) (*x < y*)
      | EQUAL => x :: y :: merge (A, B) (*x = y*)
      | GREATER => y :: merge (x :: A, B) (*x > y*)
```

$$W(n) = c_2 + 2W(n/2) + W_{\text{split}}(n) + W_{\text{merge}}(n_a, n_b) \Longrightarrow O(n \log n)$$

$$S(n) = c_2 + S(n/2) + S_{\text{split}}(n) + S_{\text{merge}}(n_a, n_b) \Longrightarrow O(n)$$

```
(* Msort(t) returns a sorted tree (including duplicates) *)
fun Msort Empty = Empty
  | Msort (Node(l, x, r)) = Ins(x, Merge(Msort l, Msort r))
fun Ins(x, Empty) = Node(Empty, x, Empty)
  | Ins(x, Node(l, y, r)) = (case Int.compare(x, y) of
     GREATER => Node(l, y, Ins(x, r))
      | _     => Node(Ins(x, l), y, r)
fun Merge(Empty, t2) = t2
  | Merge(Node(l1, x, r1), t2) =
     let val (l2, r2) = SplitAt(x, t2)
     in Node(Merge(l1, l2), x, Merge(r1, r2)) end
(* SplitAt(x, t) returns a pair (t1, t2) of sorted trees
such that t1 & t2 together contain exactly the elements of
t (incl. duplicates), the elements of t1 are LESS or equal
to x, and the elements of t2 are GREATER or equal to x *)
fun SplitAt(x, Empty) = (Empty, Empty)
  | SplitAt(x, Node(l, y, r)) = (case Int.compare(x, y) of
     LESS => let val (t1, t2) = SplitAt(x, l)
             in (t1, Node(t2, y, r)) end
      | _  => let val (t1, t2) = SplitAt(x, r)
             in (Node(l, y, t1), t2) end
```