1. 证明：For all L:int list,

msort(L) = a <-sorted permutation of L.

```
fun msort [ ] = [ ]
     | msort [x] = [x]
     | msort L =  let
          val (A, B) = split L
      in
                    merge (msort A, msort B)
      End
```

答题区域--答题区域

1. Base Case (n=0, n=1)

当 n=0 时, msort [] = []。空列表是有序的，且是自身的排列。命题成立。

当 n=1 时, msort [x] = [x]。单元素列表是有序的，且是自身的排列。命题成立。

2. Inductive Case ($n \geq 2$)

假设： 假设对于所有长度 $k < n$ 的列表 R，msort R 均是 R 的有序排列。

证明 n 的情况：

对于长度为 n 的列表 L，msort L = merge(msort A, msort B)，其中 (A, B) = split L。

根据 split 的性质, length(A) < n 且 length(B) < n。

> For all L:int list, if length(L)>1
> then split(L) = (A, B)
> where A and B have *shorter length than L*
> and A@B is a permutation of L

根据 归纳假设，msort A 是 A 的有序排列，msort B 是 B 的有序排列。

分析 msort L 的结果：

有序性 (Sorted): merge 的两个输入 msort A 和 msort B 均已有序，因此其输出 msort L 也是有序的。

排列性 (Permutation): msort L 是 (msort A) @ (msort B) 的排列 (由 merge 性质)，而 (msort A) @ (msort B) 是 A @ B 的排列 (由归纳假设)，A @ B 又是 L 的排列 (由 split 性质)。根据排列的传递性，msort L 是 L 的一个排列。

用归纳法证明Merge函数的正确性

For all <-sorted lists A and B,
merge(A, B) = a <-sorted permutation of A@B.

- **Method**: *strong* induction on length(A)*length(B).
- **Base cases**: (A, [ ]) and ([ ], B).
  (i) Show: if A is <-sorted, merge(A,[ ]) = a <-sorted perm of A@[ ].
  (ii) Show: if B is <-sorted, merge([ ],B) = a <-sorted perm of [ ]@B.
- **Inductive case**: (x::A, y::B).
  Induction Hypothesis: for all *smaller* (A′, B′), if A′ & B′ are <-sorted, merge(A′, B′) = a <-sorted perm of A′@B′.
  Show: if x::A and y::B are <-sorted,
  merge(x::A, y::B) = a <-sorted perm of (x::A)@(y::B).

因此，msort L 是 L 的一个有序排列。

3. 结论

根据归纳法原理，该命题对所有 $n \geq 0$ 均成立。

答题区域---答题区域

2. 设 P(t)表示：对所有整数 y, SplitAt(y, t) = 二元组(t1, t2)，满足

t1 中的每一项 ≤ y 且 t2 中的每一项≥ y

且 t1, t2 由 t 中元素组成

证明：对所有有序树 t，P(t)成立

```
fun SplitAt(y, Empty) = (Empty, Empty)
    | SplitAt(y, Node(t1, x, t2)) =
 case compare(x, y) of
       GREATER => let val (l1, r1) = SplitAt(y, t1)
         in   (l1, Node(r1, x, t2))
         end
    |        _         => let val (l2, r2) = SplitAt(y, t2)
         in   (Node(t1, x, l2), r2)
         end
```

定理：对所有树 t 和整数 y,

SplitAt(y, t) = 二元组(t1, t2), 满足 depth(t1) ≤ depth t 且 depth(t2) ≤ depth t

答题区域---答题区域

SplitAt 函数本质上是一个自上而下的遍历过程。它从树根开始，每到一个节点就用 y 来判断这个节点和它的子树应该往左边（≤ y）还是右边（> y）放。通过递归地调用自己，把问题分解，直到整棵树被完美地分割成两部分

1.Base Case :当 t = Empty,根据函数定义, SplitAt(y, Empty) = (Empty, Empty)。

t1 = Empty, t2 = Empty。命题 "Empty 中的每一项 $\le y$" 和 "Empty 中的每一项 $\ge y$"成立, (Empty, Empty) 的元素集合与 Empty 的元素集合相同，故命题成立。

2.Inductive Case

归纳假设 :假设对于所有 $k < n$ (其中 $n > 0$)，命题 P(k) 均成立。也就是说，对于任何深度 严格小于 n 的树 t'，SplitAt(y, t') 都能正确地将其分割。

现在我们需要证明 P(n) 也成立。

考虑一棵任意的、深度为 n ($n > 0$) 的树 t。它必然具有 Node(st1, x, st2) 的形式。

定理: 作为 t 的子树，st1 和 st2 的深度都严格小于 t 的深度 n。即 depth(st1) < n 且 depth(st2) < n。允许我们在对 st1 或 st2 进行递归调用时，应用我们的归纳假设。现在分两种程序分支讨论：

（1）情况 A: $x > y$: 函数返回 (l1, Node(r1, x, st2))，其中 (l1, r1) = SplitAt(y, st1)。由于 depth(st1) < n，我们可以对 SplitAt(y, st1) 的结果应用 归纳假设。我们得知：l1 中所有项 $\le y$，r1 中所有项 $> y$。l1 和 r1 由 st1 的元素组成。

对于 l1：根据归纳假设，其所有项 $\le y$。

对于 Node(r1, x, st2)：根 x 满足 $x > y$。

子树 r1 的所有项 $> y$ (根据归纳假设)。

子树 st2 的所有项 z 满足 $z > x$ (因原树有序)，故 $z > y$。

因此，第二部分的所有项都 $> y$。

结果的元素由 l1, r1, x, st2 组成。l1 和 r1 组成了 st1，所以总元素为 st1, x, st2，即原树 t 的全部元素，符合命题。

（2）情况 B: $x \le y$: 函数返回 (Node(st1, x, l2), r2)，其中 (l2, r2) = SplitAt(y, st2)。由于 depth(st2) < n，我们可以对 SplitAt(y, st2) 的结果应用 归纳假设。我们得知：l2 中所有项 $\le y$，r2 中所有项 $> y$。l2 和 r2 由 st2 的元素组成。

对于 Node(st1, x, l2)：根 x 满足 $x \le y$。

子树 st1 的所有项 z 满足 $z < x$ (因原树有序)，故 $z \le y$。

子树 l2 的所有项 $\le y$ (根据归纳假设)。

因此，第一部分的所有项都 $\le y$。

对于 r2：根据归纳假设，其所有项 $> y$。

结果的元素由 st1, x, l2, r2 组成。l2 和 r2 组成了 st2，所以总元素为 st1, x, st2，即原树 t 的全部元素，符合命题。

在两种情况下，对于深度为 n 的树，命题都成立。因此 P(n) 成立。

答题区域--答题区域

3. 分析以下函数或表达式的类型(不要用 smlnj)：

（1）fun all (your, base) =

case your of

0 => base

| _ => "are belong to us" :: all(your - 1, base)

your 需要支持 case your of 0 => ... 和 your - 1，所以 your : int

在 _ 分支中，函数返回 "are belong to us" :: all(your - 1, base)。

"are belong to us" 是一个 string。

:: 是列表的 cons 操作符，它将一个元素添加到列表的头部。

因此，all(your - 1, base) 的返回值必须是一个列表，并且列表中的元素类型必须是 string。也就是说，函数的返回类型必须是 string list

所以最终类型为 val all = fn : int * string list -> string list

（2）fun funny (f, []) = 0

| funny (f, x::xs) = f(x, funny(f, xs))

当列表为空时，返回 0，所以 funny 整体返回类型必须是 int

funny (f, x::xs) 调用 f(x, funny(f, xs))

f 接受一个 'a 类型(列表的元素类型）和一个 int 类型的参数，并返回一个 int。

所以 f 的类型是 'a * int -> int

funny = fn : ('a * int -> int) * 'a list -> int

（3）(fn x => (fn y => x)) "Hello, World!"

它接受一个参数 x。我们称 x 的类型为 'a。返回另一个匿名函数 fn y => x。

这个内部函数接受一个参数 y（我们称其类型为 'b），返回外部函数捕获的 x。类型是 'a。所以，内部函数的类型是 'b -> 'a。

因此，外部函数 fn x => ... 的类型是：接受一个 'a 类型的参数，返回一个 'b -> 'a 类型的函数。其类型为 'a -> 'b -> 'a

参数 "Hello, World!" 的类型是 string。

SML 会将参数类型 string 与函数第一个参数的类型 'a 进行合一，因此 'a 被确定为 string。函数应用的结果是返回内部函数，此时类型变量 'a 已被具体化。

返回的函数是 fn y => "Hello, World!"。

这个返回的函数的类型是 'b -> string，因为 y 的类型 'b 仍然是泛型。

4. 给定一个数组 A[1..n]，前缀和数组 PrefixSum[1..n]定义为： PrefixSum[i] = A[0]+A[1]+...+A[i-1];

例如：PrefixSum [ ] = [ ]

PrefixSum [5,4,2] = [5, 9, 11]

PrefixSum [5,6,7,8] = [5,11,18,26]

试编写：

(2)函数 PrefixSum: int list -> int list，

要求：PrefixSum(n) = O(n2)。(n 为输入 int list 的长度)

(2) 函数 fastPrefixSum: int list -> int list，

要求： fastPrefixSum(n) =O(n).

（提示：可借助帮助函数 PrefixSumHelp）

```
(*****Begin*****)
(* === O(n^2) 版本：PrefixSum ===
   思路：
     对于结果中第 i 个位置（i 从 1 开始计），
     我们重新把输入列表的前 i 个元素取出来，然后求和。
     由于每个位置都重新计算前缀和，整体工作量为
     sum_{i=1..n} O(i) = O(n^2)。
*)
fun PrefixSum lst =
  let
    (* prefix i: 计算输入列表 lst 的前 i 个元素之和
       List.take(lst, i) 返回 lst 的前 i 个元素（如果 i > 长度则取全部）。
       List.foldl (op +) 0 xs 把 xs 中元素累加（从左向右），初始值为 0。 *)
    fun prefix i =
      List.foldl (op +) 0 (List.take(lst, i))
    (* loop (i, n): 递归构造结果列表，从 i = 1 到 n。
       当 i > n 时返回空列表；否则把 prefix i 加到结果前面。 *)
    fun loop (i, n) =
      if i > n then []
      else prefix i :: loop (i+1, n)
  in
    loop (1, List.length lst)  (* 从 1 开始到列表长度 *)
  end;
(* === O(n) 版本：fastPrefixSum ===
   思路：
     只遍历列表一次，维护一个累加器 sum，
     对每个元素 x 把 sum + x 作为当前前缀和加入结果。
     为了高效（不在每步重复扫描前缀），这是线性时间的做法。
     使用尾递归并且最后对结果反转以保持原顺序。
*)
fun fastPrefixSum lst =
  let
    (* helper (restList, currentSum, accList)
```

```
       - restList: 未处理的输入部分
       - currentSum: 已处理部分的累计和（在处理 x 之前的和）
       - accList: 已产生的前缀和（**逆序**存放，便于用 :: 高效添加）
       返回最终正确顺序的前缀和列表。 *)
    fun helper ([], _, acc) = List.rev acc
      | helper (x::xs, sum, acc) =
          let
            val newSum = sum + x
          in
            (* 在 acc 前面加入 newSum，然后继续递归（尾递归） *)
            helper (xs, newSum, newSum::acc)
          end
  in
    helper (lst, 0, [])
  end;
(*****End*****)
```

自测运行结果

5. 一棵 minheap 树定义为：

1. t is Empty;

2. t is a Node(L, x, R), where R, L are minheaps and values(L), value(R) >= x (value(T)函数用于获取树 T 的根节点的值）

编写函数 treecompare, SwapDown 和 heapify：

```
treecompare: tree * tree -> order
(* when given two trees, returns a value of type order, based on which
tree has a larger
value at the root node *)
SwapDown: tree -> tree
(* REQUIRES the subtrees of t are both minheaps
* ENSURES swapDown(t) = if t is Empty or all of t’s immediate children
are empty then
* just return t, otherwise returns a minheap which contains exactly the
 elements in t. *)
heapify : tree -> tree
```

```
(* given an arbitrary tree t, evaluates to a minheap with exactly the e
lements of t.  *)
```

分析 SwapDown 和 heapify 两个函数的 work 和 span。

答题区域---答题区域

题目分析：

1：最小堆定义

minheap 树（最小堆） 是一种二叉树，满足以下条件：

如果树是空树（Empty），那么它是一个 minheap。

如果树是 Node(L, x, R)，其中：

L 和 R 都是 minheap；

$x \le value(L)$ 且 $x \le value(R)$，即父节点的值不大于两个子树的根值。

换句话说，minheap 的根节点值最小，堆序性质在整棵树中成立。

2.函数功能和实现

```
(* 树的数据类型定义 *)
datatype tree = Empty
              | Node of tree * int * tree
(* 序关系类型 *)
datatype order = LESS | EQUAL | GREATER
(* 辅助函数：获取树的根节点值 *)
(* 如果是空树，返回一个很大的值（表示无穷大） *)
fun value(Empty) = valOf(Int.maxInt)
  | value(Node(_, x, _)) = x
```

（1）treecompare

比较两棵树根节点的值，返回一个 order 类型结果：

如果 $value(t1) < value(t2)$，返回 LESS；

如果 $value(t1) > value(t2)$，返回 GREATER；

如果相等，返回 EQUAL。

```
fun treecompare(t1: tree, t2: tree): order =
    let
        val v1 = value(t1)
        val v2 = value(t2)
    in
        if v1 < v2 then LESS
        else if v1 > v2 then GREATER
        else EQUAL
    end
```

（2） SwapDown

通过”下沉”操作修复堆性质

前提是输入树的左右子树本身都是 minheap；

如果根节点比左右孩子大，就要把根节点向下交换，直到恢复 minheap 性质。

如果树为空，或者没有子节点，就直接返回。

结果是一个包含相同元素的 minheap。

```
fun swapDown(t: tree): tree =
    case t of
        Empty => Empty
      | Node(Empty, x, Empty) => t  (* 叶子节点，无需交换 *)
      | Node(L, x, Empty) =>          (* 只有左子树 *)
            if value(L) < x then
                (* 左子节点更小，需要交换 *)
                (case L of
                    Node(LL, y, LR) => Node(Node(LL, x, LR), y, Empty)
                  | Empty => t)  (* 不会发生 *)
            else t
      | Node(Empty, x, R) =>          (* 只有右子树 *)
            if value(R) < x then
                (* 右子节点更小，需要交换 *)
                (case R of
                    Node(RL, y, RR) => Node(Empty, y, Node(RL, x, RR))
                  | Empty => t)  (* 不会发生 *)
            else t
      | Node(L, x, R) =>              (* 有两个子树 *)
            let
                val vL = value(L)
                val vR = value(R)
            in
                (* 找出较小的子节点 *)
                if x <= vL andalso x <= vR then
                    t  (* 根节点已经是最小的，无需交换 *)
                else if vL <= vR then
                    (* 左子节点更小，与左子树的根交换 *)
                    (case L of
                        Node(LL, y, LR) => Node(Node(LL, x, LR), y, R)
                      | Empty => t)  (* 不会发生 *)
                else
                    (* 右子节点更小，与右子树的根交换 *)
                    (case R of
                        Node(RL, y, RR) => Node(L, y, Node(RL, x, RR))
                      | Empty => t)  (* 不会发生 *)
```

```
            end
```

（3）heapify

将任意一棵树转换成一个 minheap，保持所有元素不变。

方法：对整棵树的左右子树先递归调用 heapify，然后对根节点调用 SwapDown，保证堆序。

```
fun heapify(t: tree): tree =
    case t of
        Empty => Empty
      | Node(L, x, R) =>
            let
                (* 递归地将左右子树转换为 MinHeap *)
                val heapL = heapify(L)
                val heapR = heapify(R)
                (* 构造新树并执行 swapDown *)
                val newTree = Node(heapL, x, heapR)
            in
                swapDown(newTree)
            end
```

3.性能分析

Work (W): 总工作量，即所有处理器执行的总操作数，相当于串行执行时间。

Span (S): 跨度，即最长依赖路径的长度，相当于在拥有无限处理器的情况下执行所需的时间。

（1)SwapDown 复杂度分析

SwapDown 的执行路径是线性的。在每一步，它进行常数次比较和操作，然后最多对一个子树进行一次递归调用。

Work (W): 每次递归调用处理向下一层，工作量为 O(1)。这个过程最多持续 d 次（树的深度）。

Work = O(d)

Span (S): 所有操作都是顺序的，没有可并行化的部分

Span = O(d)

（2） heapify 复杂度分析

heapify 在每个节点上执行操作。它首先处理子树，然后处理当前节点。

Work (W): heapify 访问树中的每个节点一次。对每个节点，heapify 都会：

递归处理左右子树

调用一次 swapDown（O(d)）

递推关系：W(n) = W(左子树) + W(右子树) + O(d)

对于完全二叉树，这个递推求解得到：Work = O(n)

Span (S): 左右子树的 heapify 可以并行执行

假设树大致平衡，则 d_L 和 d_R 约等于 d-1。

S(d) ≈ S(d-1) + O(d)

展开这个递推式：S(d) = O(d) + O(d-1) + O(d-2) + ... + O(1)

这是前 d 个整数的和，所以：Span = O(d^2)

对于平衡树 d 约为 O(log n)，heapify 的 Span 则为 O((log n)^2),

SwapDown：Work = O(d)，Span = O(d)

heapify：Work = O(n)，Span = O(d^2)