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
```

解答：我们根据 L 的长度用归纳法进行证明：

（1）L = [ ], [x]时

- msort [ ]的值为[ ]，这是 L（即[ ]）的一个有序排列
- msort [x]的值为[x]，这是 L（即[x]）的一个有序排列

（2）假设任何长度小于 k 的 list L‘都有 msort L’是一个有序排列；我们要证明，对于长度为 k 的 list L，也要有 msort L 是一个有序排列。

- 首先 mosrt L 函数用 split 将 L 分成 A、B 的两部分，这两部分长度小于 L 的长度。
- 由假设得 msort A 和 msort B 都是有序排列。
- 故由 merge 函数的性质，两个有序排列的 merge 是有序排列，故 merge(msort A, msort B)是有序排列。
- 故最后我们得到的排列是有序的，且这个排列中元素为 A@B =L

综上，msort L 能够获得 L 的有序排列。

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

---

解答：我们采用数的结构归纳法解决这个问题：

（1） Base case：当树为空时(t = Empty)，(t1, t2) = (Empty, Empty)。由于 t1 和 t2 中都没有元素，所以不存在 t1 中有元素不满足小于等于 y，或者 t2 中有元素不满足大于等于 y 的情况出现。既然 t1 和 t2 中没有不满足题目条件的元素，又由于 t1、t2 肯定是由 t 中的元素组成（因为 t1、t2 和 t 都是空），故 P(t) 成立。

（2） Inductive case：当树 t = Node(t1, x, t2)的时候，假设 P(t1)和 P(t2)都成立（即树 t1 和 t2 被分割对于其左右子树有如上结论），此时我们要证明 P(Node(t1, x, t2))成立。

下面是证明的过程：我们比较 x 和 y，分成大于和小于等于两种情况：

- 如果 x 大于 y，我们得到的二元组是(l1, Node(r1, x, t2))。由于我们假设了 P(t1)

成立，而 l1 是用 Split(y, t1)形成的左子树，故 l1 中的每一项都小于等于 y；r1 是 Split(y, t1)形成的右子树，故 r1 中的每一项都大于等于 y。由于树是有序树，而 x 大于 y，故 t2 中所有的元素也是大于 y 的。综上，(l1, Node(r1, x, r2))中 l1 的每一项都小于等于 y，Node(r1, x, t2)每一项（r1、x、t2）都大于等于 y。

- 如果 x 小于等于 y，我们得到的二元组是(Node(t1, x, l2), r2)。由于我们假设了 P(t2)成立，而 r2 是用 Split(y, t2)形成的右子树，则 r2 中的每一个值都大于等于 y；l2 是用 Split(y, t2)形成的左子树，故 l2 中的每一项都小于等于 y。由于树是有序树，而 x 小于等于 y，则 t1 中的每一项都小于等于 y。综上，(Node(t1, x, l2), r2)中 r2 的每一项都大于等于 y，Node(t1, x, l2)中每一项（t1、x、l2）都小于等于 y。

综上，对于所有有序树 t，都有 P(t)成立。

3. 分析以下函数或表达式的类型(不要用 smlnj)：

```
fun all (your, base) =
        case your of
              0 => base
            | _ => "are belong to us" :: all(your - 1, base)

fun funny (f, []) = 0
    | funny (f, x::xs) = f(x, funny(f, xs))

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

解答：分析如下：

（1） 第一个函数 all(your, base)

该函数的类型为 int * string list -> string list

- 首先我们分析 case your of 0 这一句话，证明 your 和 0 是一个类型，即 int 型。
- 再观察“are belong to us”:: all (your – 1, base)这句话，可以知道函数的返回值为一个 string list，即不停的在 list 中加入 string。
- 再观察 case 0 => base 这一句话，可以知道 base 和函数返回值为一个类型，都是 string list。
- 综上，该函数的类型为 int * string list -> string list

（2） 第二个函数 funny(f, [ ])

该函数的类型为(‘a * int-> int) * ‘a list-> int

- 首先观察函数的返回值 funny(f, [ ]) = 0 可以判断返回值类型为 int
- 通过参数值(f, [ ])我们可以判断参数类型为 func * ‘a list，下面我们需要判断这个 func 的类型。
- 通过 funny (f, x :: xs) = f(x, funny(f, xs))可以判断 f 的返回值和 funny 一样为 int 型，f 的参数类型为 ‘a * int。
- 综上，该函数的类型为(‘a * int -> int) * ‘a list -> int

（3） 第三个表达式(fn x => (fn y => x)) “Hello, World!”

该函数的类型为'a -> string

- 首先我们看出(fn x => (fn y => x))的类型应该是 'a -> 'b ->'a。
- 然后我们赋值"Hello, World!"将'a 的类型固定为 string。所以我们能够得到一个新类型'a -> string

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：

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

要求：$W_{PrefixSum}(n) = O(n^2)$。(n 为输入 int list 的长度)

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

要求： $W_{fastPrefixSum}(n) = O(n)$.

（提示：可借助帮助函数 PrefixSumHelp）

解答：函数构造过程以及结果测试如下：

（1） Prefixsum 函数的构造

这个函数要求我们的 work 为 $O(n^2)$，我们能够想到的方法是，将原来 list 中的每个元素都叠加到后面的元素上去。这样每加一次元素，都需要遍历整个 list（比当前元素下标大的部分），这样每一次都是 O(n)，一共遍历 n 次，所以总 work 是 $O(n^2)$。具体实现方法如下：

- 首先我们构造函数 ListAddx(L, x)，即将 x 加到 list 中的每一个元素上。这个函数当 list 为空的时候返回空，当 list 不为空的时候，取出其第一个元素和 x 相加，返回到结果 list 的第一个元素，剩下的 list 部分继续调用 ListAddx 函数与 x 相加。
- 接着我们在 PrefixSum 函数中调用 ListAddx 函数实现功能。如果输入参数为空，则我们返回空；如果输入参数不为空 list，则我们取出该 list 的首元素，返回到结果 list 的第一元素，剩余的 list 部分递归调用 Prefixsum 处理完整个 list，然后再调用 ListAddx 将处理完的 list 与相加。

函数代码如下：

```
(* question 4 *)
(* ListAddx: int list * int -> int list *)
(* PrefixSum: int list -> int list *)
(* 要求: WPrefixSum(n) = O(n2)。(n为输入int list的长度) *)
fun ListAddx ([ ], x) = [ ]
    | ListAddx (y :: L, x) = (x + y) :: ListAddx (L, x)
fun PrefixSum [ ] = [ ]
    | PrefixSum (x :: L) = x :: ListAddx (PrefixSum L, x)
```

（2） fastPrefixSum 函数的构造

这个函数要求我们只用 O(n)的 work 实现前序相加的功能。这里我们需要借助帮助函数 PrefixSumHelp，实现尾递归，将前面相加好的结果作为参数传递，后面只需要加前面相加好的结果和即可，不需要像上一个函数一样依次去加每个量。这样，我们只需要遍历一遍就能够完成整个前序和的相加，其 work 为 O(n)。具体实现思路如下：

- 首先构造 PrefixSumHelp(L, x)函数实现尾递归。如果 list 为空，则返回空；如果 list 不为空，取出其第一个元素（假设这个元素是 y）与 x 相加，返回到结果 list 的第一个元素；剩下的部分继续调用 PreSumHelp 函数与 x+y 相加。
- 可以看出 PrefixSumHelp(L, x)和上面一种方法 ListAddx(L, x)的唯一区别是在递归的时候我们用 x+y 而不是 x 与剩下的表相加。
- 接下来构造函数 fasttPrefixSum L，我们直接调用 PrefixSumHelp(L, x)即可，需要注意的是设定 x 的初始状态为 0

函数代码如下：

```
(* PrefixSumHelp: int list * int -> int list *)
(* fastPrefixSum: int list -> int list *)
(* 要求: WfastPrefixSum(n) =O(n).  *)
fun PrefixSumHelp ([ ], x) = [ ]
    | PrefixSumHelp (y :: L, x) = (x + y) :: PrefixSumHelp (L, x + y)

fun fastPrefixSum L = PrefixSumHelp (L, 0)
```

（3） 两个函数的测试

测试截图如下：

```
(* test_question 4 *)
val result1 = PrefixSum [1,2,3]      (* [1,3,6] *)
val result2 = PrefixSum [6,3,1]      (* [6,9,10] *)

val result3 = fastPrefixSum [1,2,3]  (* [1,3,6] *)
val result4 = fastPrefixSum [6,3,1]  (* [6,9,10] *)
```

```
val ListAddx = fn : int list * int -> int list
val PrefixSum = fn : int list -> int list
val PrefixSumHelp = fn : int list * int -> int list
val fastPrefixSum = fn : int list -> int list
val result1 = [1,3,6] : int list
val result2 = [6,9,10] : int list
val result3 = [1,3,6] : int list
val result4 = [6,9,10] : int list
```

测试的解释：

- result1 和 result2 是测试 PrefixSum 函数，可以看出结果和后面预期（注释部分）一致，函数正确。
- result1 和 result2 是测试 fastPrefixSum 函数，采用了和上面一样的测试样例，可以看出结果和后面预期（注释部分）一致，函数正确。

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 elements of *t*. *)

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

（1） treecompare 的实现：

函数主要的实现功能是比较两棵树的根节点的大小。

- 如果两棵树都为空，则为 EQUAL。
- 如果一棵树为空一棵树，则不为空的那一棵树大于为空的那一棵。
- 如果两棵树都不为 Empty，则比较其根节点的大小。

代码如下图所示：

```
(* question 5 *)
(* defination of tree*)
datatype tree = Empty | Node of tree * int * tree
(* 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 *)
fun treecompare (Empty, Empty) = EQUAL
    | treecompare (Empty, _ ) = LESS
    | treecompare ( _ , Empty) = GREATER
    | treecompare (Node( _ , i , _ ), Node( _ , j , _ )) = Int.compare(i, j)
```

（2） SwapDown 的实现：

函数的主要功能是在保证根节点的左右子树都是 minheap 的情况下，构造一棵 minheap。这个函数的麻烦之处就是我们无法确定总的根节点与左右子树的大小关系，以及其该如何处理。所以我们先要构造一个比较函数，比较总根节点与左子树或者右子树的根节点的大小。我们设要比较的根节点为 x，要比较的子树为 Node(l, y, r)，比较之后返回三元组(x 是否大于 y, 当前 root, 新构成的子树)，函数名为 SwapDown 具体如下：

- 如果要比较的子树为空，则返回三元组(false, x, Empty)
- 如果要比较的子树不为空，如果 x 大于 y，则更新 root 为 y，x 作为根节点作为原来的子树，返回三元组(true, y, Node(l, x, r))；如果是 x 小于等于 y，则不更新 root 也不更新子树，返回三元组(false, x, Node(l, y, r))。

在实现了 cmproot 的情况下，我们实现 SwapDown，只需要对根节点大小关系不符合

的那一边递归使用 SwapDown 函数即可。

- 当输入树为 Empty 的时候，返回 Empty。
- 输入树为只有一个根节点的时候，返回原树。
- 其他情况下，设这棵树为 Node(l, x, r)，调用 cmproot 函数比较根节点 x 和左子树根节点值以及根节点 x 和右子树根节点值。获得两个三元组(cmpleft, root1, left)、（cmpright, root2, right）,如果 cmpleft 为真，则将新树的根节点换成 root1，右子树为 r，左子树 left 递归调用 SwapDown，然后再对新树调用 SwapDown。
- 否则，如果 cmpright 为真，则将新树的根节点换成 root2，左子树为 l，右子树 right 递归调用 SwapDown，然后再对新树调用 SwapDown。
- 如果 cmpleft 和 cmpright 都不为真，则直接返回(l, x, r)。

（3） heapify 的实现：

函数的主要功能是将任意一棵树转换为 minheap。在实现了上述 SwapDown 函数后这个函数遍很容易实现了，只需要调用 SwapDwon 并在左右子树递归调用 heapify 即可。

- 当要处理的树为空的时候，就返回空，
- 其他情况下，将左右子树递归调用 heapify 函数，这样左右子树就是 minheap 了，然后再将 heapify 之后的左右子树结果和根节点一起，调用 SwapDown，得到了整棵 minheap。

代码如下图所示：

```
(* heapify : tree -> tree *)
(* given an arbitrary tree t, evaluates to a minheap with exactly the elements of t. *)
fun heapify Empty = Empty
    | heapify (Node (l, x, r)) = SwapDown (Node (heapify l, x, heapify r))
```

（4） 三个函数的测试：

```
(* test_question 5 *)
val tree1 = Empty
val tree2 = Node(Node(Empty, 1, Empty), 2, Node(Empty, 3, Empty))
val tree3 = Node(Node(Empty, 7, Node(Empty, 4, Empty)), 6, Node(Empty, 5, Empty))

val result5 = treecompare(tree1, Empty)                  (* EQUAL *)
val result6 = treecompare(tree2, Empty)                  (* GREATER *)
val result7 = treecompare(tree2, Node(Empty, 2, Empty))  (* EQUAL *)

val result8 = SwapDown tree1 (* Empty *)
val result9 = SwapDown tree2 (* Node(Node(Empty,2,Empty),1,Node(Empty,3,Empty)) *)

val result10 = heapify tree1 (* Empty *)
val result11 = heapify tree2 (* Node(Node(Empty,2,Empty),1,Node(Empty,3,Empty)) *)
val result12 = heapify tree3 (* Node(Node(Empty,6,Node(Empty,7,Empty)),5,Node(Empty,6,Empty)) *)
```

```
val treecompare = fn : tree * tree -> order
val cmproot = fn : int * tree -> bool * int * tree
val SwapDown = fn : tree -> tree
val heapify = fn : tree -> tree
val tree1 = Empty : tree
val tree2 = Node (Node (Empty,1,Empty),2,Node (Empty,3,Empty)) : tree
val tree3 = Node (Node (Empty,7,Node #),6,Node (Empty,5,Empty)) : tree
val result5 = EQUAL : order
val result6 = GREATER : order
val result7 = EQUAL : order
val result8 = Empty : tree
val result9 = Node (Node (Empty,2,Empty),1,Node (Empty,3,Empty)) : tree
val result10 = Empty : tree
val result11 = Node (Node (Empty,2,Empty),1,Node (Empty,3,Empty)) : tree
val result12 = Node (Node (Empty,6,Node #),4,Node (Empty,5,Empty)) : tree
val it = () : unit
```

测试的解释：

- tree1、tree2、tree3 是三棵测试用的树。
- result5、result6、result7 是测试 treecompare 函数，可以看出结果和后面预期（注释部分）一致，函数正确。
- result8、result9 是测试 SwapDown 函数，可以看出结果和后面预期（注释部分）一致，函数正确。

- result10、result11、result12 是测试 heapify 函数，可以看出结果和后面预期（注释部分）一致，函数正确。

（5） 分析 SwapDown 和 heapify 两个函数的 work 和 span

我们设树的深度为 d，下面我们对于深度为 d 的树，分析两个函数的 work 和 span。

- 对于 SwapDown 来说：
  - ✓ 先分析它的 work：

    $W_{SwapDown}(0) = c_0$

    $W_{SwapDown}(d) = c_1 + W_{SwapDown}(d-1)$

    即我们可以通过递推得到：

    $W_{SwapDown}(d) = c_0 + d * c_1 = O(d)$
  - ✓ 分析它的 span：

    $S_{SwapDown}(0) = c_0$

    $S_{SwapDown}(d) = c_1 + S_{SwapDown}(d-1)$

    即我们可以通过递推得到：

    $S_{SwapDown}(d) = c_0 + d * c_1 = O(d)$

- 对于 heapify 来说：
  - ✓ 先分析它的 work：

    $W_{heapify}(0) = c_0$

    $W_{heapify}(d) = c_1 + 2 * W_{heapify}(d-1) + W_{SwapDown}(d)$

    由于我们已经计算出$W_{SwapDown}(d) = O(d)$，所以我们有：

    $W_{heapify}(d) = 2 * W_{heapify}(d-1) + O(d) = (1 + 2 + 4 + \cdots + 2^d) * O(d)$

    $= (2^{d+1} - 1) * O(d) = O(d2^d)$

    $W_{heapify}(d) = O(d2^d)$
  - ✓ 分析它的 span：

    $S_{heapify}(0) = c_0$

    $S_{heapify}(d) = c_1 + S_{heapify}(d-1) + S_{SwapDown}(d)$

    由于我们已经算出：$S_{SwapDown}(d) = O(d)$，故我们有：

    $S_{heapify}(d) = S_{heapify}(d-1) + O(d) = d * O(d) = O(d^2)$