Create a type for pairs of numbers.

```
Inductive natprod : Type :=
  pair : nat -> nat -> natprod.
```

Define functions to extract members of pairs of numbers.

```
Definition fst (p : natprod) : nat :=
  match p with
  | (x,y) => x
  end.
Definition snd (p : natprod) : nat :=
  match p with
  | (x,y) => y
  end.
```

State that the `pair` constructor is surjective.

```
Theorem surjective_pairing : forall (n m : nat),
  (n,m) = (fst (n,m), snd (n,m)).
```

Create a type for a list of numbers.

```
Inductive natlist : Type :=
  | nil : natlist
  | cons : nat -> natlist -> natlist.
```

Define the length of a nat list.

```
Fixpoint length (l:natlist) : nat :=
  match l with
  | nil => O
  | h :: t => S (length t)
  end.
```

Define an append function for nat lists.

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil    => l2
  | h :: t => h :: (app t l2)
  end.
```

Define a subset function for bags.

```
Fixpoint subset (s1:bag) (s2:bag) : bool :=
  match s1 with
  | nil    => true
  | h :: t => match member h s2 with
              | false => false
              | true  => subset t (remove_one h s2)
              end
  end.
```

State the associativity of nat lists.

```
Theorem app_ass : forall l1 l2 l3 : natlist,
  (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
```

Define a snoc function for nat lists.

```
Fixpoint snoc (l:natlist) (v:nat) : natlist :=
  match l with
  | nil    => [v]
  | h :: t => h :: (snoc t v)
  end.
```

Define a reverse function for nat lists.

```
Fixpoint rev (l:natlist) : natlist :=
  match l with
  | nil    => nil
  | h :: t => snoc (rev t) h
  end.
```

State that the empty nat list is the right identity with respect to append.

```
Theorem app_nil_end : forall l : natlist,
  l ++ [] = l.
```

State a fundamental relationship between `rev` and `snoc`.

```
Theorem rev_snoc : forall (l:natlist) (n:nat),
  rev(snoc l n) = n :: rev l.
```

State that reversing a nat list is involutive.

```
Theorem rev_involutive : forall l : natlist,
  rev (rev l) = l.
```

Create an option time for nat lists.

```
Inductive natoption : Type :=
  | Some : nat -> natoption
  | None : natoption.
```

Define an index function for nat lists.

```
Fixpoint index (n:nat) (l:natlist) : natoption :=
  match l with
  | nil => None
  | a :: l' => match beq_nat n O with
               | true => Some a
               | false => index (pred n) l'
               end
  end.
```

State that equality of nats is symmetric.

```
Theorem beq_nat_sym : forall (n m : nat),
  beq_nat n m = beq_nat m n.
```

Create a dictionary type for nats.

```
Inductive dictionary : Type :=
  | empty  : dictionary
  | record : nat -> nat -> dictionary -> dictionary.
```

Define key/value insertion for nat dictionaries.

```
Definition insert (key value : nat) (d : dictionary) : dictionary :=
  (record key value d).
```

Define a lookup function for nat dictionaries.

```
Fixpoint find (key : nat) (d : dictionary) : option nat :=
  match d with
  | empty         => None
  | record k v d' => if (beq_nat key k) then (Some v) else (find key d')
  end.
```

State the first important dictionary invariant.

```
Theorem dictionary_invariant1 : forall (d : dictionary) (k v: nat),
  (find k (insert k v d)) = Some v.
```

State the second important dictionary invariant.

```
Theorem dictionary_invariant2 : forall (d : dictionary) (m n o: nat),
  (beq_nat m n) = false -> (find m d) = (find m (insert n o d)).
```