⟨*neckl.py*⟩≡
```
from regina import *
```

Necklace gluings with n beads

Definition 0.1. An n-dipyramid is a 3-ball with a cell structure whose boundary triangulation is the suspension of an n-gon. The choice of suspension points is canonical unless $n = 4$, in which case we take care to remember which vertices are suspension points.

The following is our implementation of this definition in Regina.

⟨*how to make an n-dipyr*⟩≡
```
def make_dipyr(n):
        """Returns an n-dipyr."""
        newt = NTriangulation()
        for i in range(0,n):
                newt.newTetrahedron()
        for i in range(0,n):
                me = newt.getTetrahedron(i)
                you = newt.getTetrahedron((i+1)%n)
                me.joinTo(2,you,NPerm4(2,3))
        return newt
```

⟨*neckl.py*⟩+≡
```
⟨how to make an n-dipyr⟩
```

Definition 0.2. A *necklace gluing of bead number n* is a oriented face-pairing of all the faces of an n-dipyramid such that every face-pair preserves the partition of the dipyramid's vertices into suspension points and "lateral" points.

The set of necklace gluings of bead number n on an n-dipyramid is naturally in bijection with the fixed-point-free involutions on the set of faces of that n-dipyramid. There are $2 \cdot n$ faces, so given any labelling of the faces by naturals less than $2 \cdot n$, we get an identification sending fixed-point-free involutions in $S_{2 \cdot n}$ to necklace gluings.

The labelling we choose involves a choice of higher and lower suspension point, an orientation of the base polygon, and a choice of face thereon. With these choices, mark the sides of the base polygon with naturals between 0 and n, such that the chosen face takes marking 0, and the cyclic ordering on faces from the orientation induces the cyclic ordering coming from the quotient monoid $\mathbf{N}/n\mathbf{N}$. Then a face of the n-dipyramid which is adjacent to the lower suspension point takes as a label the marking on the side of the base of which it is a cone, but a face of the n-dipyramid adjacent to the higher suspension point takes instead n plus the marking on its base side. (Cf. Figure 1.)

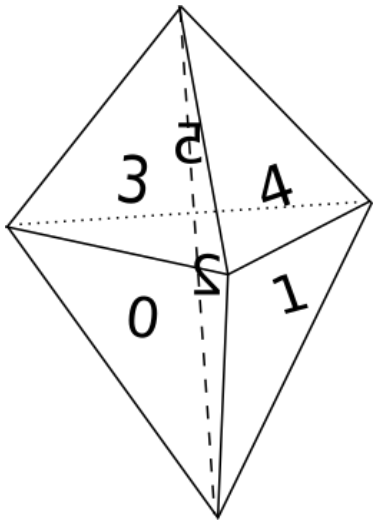

Figure 1: Our labelled 3-dipyramid.

As you can verify, we have glued up the n-dipyramid so that the interior faces are all faces 2 and 3 of their respective tetrahedra, and the boundary faces are thus faces 0 and 1. Their common edge in a given tetrahedron is an edge of the base polygon. We have also indexed the tetrahedra with naturals less than n so that the induced cyclic ordering on the common edges is the same as that induced by the base polygon. Finally, as you may verify, all the 0 faces lie around one suspension point, and the 1 faces lie around the other suspension point.

N.B. In the above paragraph, the labels 0,1,2,3 refer to Regina's internal labelling system, not our just imposed face labelling on the boundary faces of the dipyramid.

Therefore, the following code accomplishes a necklace gluing, given a fixed-point-free involution in $S_{2\cdot n}$ in cycle notation, assuming Regina maintains the indexing of tetrahedra.

⟨*how to make a necklace gluing*⟩≡
```
  def which_tet(n,face):
      if face < n:
          return face
      else:
          return face - n

  def make_necklace(fpfinv):
      n = len(fpfinv)
      ndipyr = make_dipyr(n)
      local = list(fpfinv)
      while local != []:
          (i,j) = local.pop()
          me = ndipyr.getTetrahedron(which_tet(n,i))
          you = ndipyr.getTetrahedron(which_tet(n,j))
```

⟨*neckl.py*⟩+≡
```
  ⟨how to make a necklace gluing⟩
```

The face of `me` getting glued is either 0 or 1 according, respectively, as i is less than or at least n.

⟨*how to make a necklace gluing*⟩+≡
```
          if i < n:
                  f = 0
          else:
                  f = 1
```

We now split into two cases according as i, j lie on the same or different suspension points.

If they lie along the same suspension point, then the face-pairing restricted to the vertices fixes 0 and 1. (Recall the duality between faces and vertices of a tetrahedron.) But it must reverse orientation, so that the whole manifold is oriented. Thus the induced map on the vertices is the permutation (2 3).

⟨*how to make a necklace gluing*⟩+≡
```
          if ((i < n) == (j < n)):
              me.joinTo(f,you,NPerm4(2,3))
```

On the other hand, if they lie along different suspension points, then the 0,1 vertices must be flipped. Thus the 2 and 3 vertices must be fixed so that the gluing is oriented.

⟨*how to make a necklace gluing*⟩+≡
```
          else:
              me.joinTo(f,you,NPerm4(0,1))
      ndipyr.setPacketLabel(str(fpfinv))
      return ndipyr
```

Enumerating necklace gluings Our next goal is to get a list as small as possible which for every necklace gluing G contains a manifold homeomorphic to G. We could, of course, just enumerate all fixed-point-free involutions and put the associated gluings in a list. But many of these, presumably, will be asymmetric, so that by applying elements of the symmetry group of the n-dipyr, we get different gluings which are homeomorphic. We wish to eliminate as much of such redundancy as we reasonably can.

A reasonable approach is to have our list contain one representative from each dipyr-symmetry group orbit of fixed-point-free involutions, and nothing else. Milley took this approach, and it is the approach we take as well. However, we introduce a subtlety which makes the enumeration faster and leaner.

We do not begin by enumerating all fixed-point-free involutions as Milley does. Instead, we begin by constructing a tree with a face-pair at each internal node, such that there is an identification of fixed-point-free involutions in $S_{2\cdot n}$ with backtrack-free paths from the root of the tree to leaves. This takes much less space to store than the whole list of fixed-point-free involutions. We define what it means to remove an involution from the tree. This takes much less time than removing an involution from a list. Finally, we do the most natural thing after that: let R be an empty list to contain representatives, and then while the tree still contains an involution σ, put σ in R, and remove every element of σ's orbit from the tree. Return R, the desired list of orbit representatives.

Let us call the dipyramidal symmetry group DP. Our first order of business is encoding its action on face-pairings. It also acts on face-pairs and, of course, on faces. Since it acts on several things at once, we have different functions associated to these actions. But let us begin with how DP acts on faces.

A face we represent as a natural number below $2 \cdot n$. There are three symmetries of the dipyramid which generate its symmetry group and are relatively easy to write down: a counterclockwise rotation by an nth of a revolution; the reflection through the plane through the base, which we call a *p-flip*; and a reflection through a plane through the suspension points and a lateral vertex, which we call a *v-flip*. For the lateral vertex, we pick the vertex common to the faces labelled $0, n-1, n, 2\cdot n-1$.

We contend the following implements these symmetries as they act on face labels. For convenience, we implement the rotation through t $n$ths of a revolution.

⟨*how to act*⟩≡
```
  def rt(n,t,face):
      if face < n:
          if face + t < n:
              return face + t
          else:
              return face + t - n
      else:
          if face + t < 2 * n:
              return face + t
          else:
              return face + t - n

  def p_fl(n,face):
      if face < n:
          return face + n
      else:
          return face - n

  def v_fl(n,face):
      if face < n:
          return n - (face + 1)
      else:
          return (3 * n) - (face + 1)
```

⟨*neckl.py*⟩+≡
```
  ⟨how to act⟩
```

You may check that a symmetry of the dipyramid is a rotation followed possibly by the p-flip, followed possibly by the v-flip.

Because DP acts on so many things, we introduce a poor man's datatype to represent its elements. We represent an element of DP by a pair `(s,t)` of a string and number, where the string `str` must be one of `"Rot"`, `"PRot"`, `"VRot"`, or `"VPRot"`, and the number `t` is assumed less than n.

⟨*how to act*⟩+≡
```
  def act_face (n,dp,face):
      (s,t) = dp
      r = rt(n,t,face)
      if s == "Rot":
          return r
      if s == "VRot":
          return v_fl(n,r)
      if s == "PRot":
          return p_fl(n,r)
      if s == "VPRot":
          return v_fl(n,p_fl(n,r))
      else:
          raise Exception
```

Now, to discuss the action on face-pairs, we must discuss our representation of face-pairs. We represent them not just as ordered pairs, but as consistently ordered pairs. We represent the pair of faces i, j by (i, j) or (j, i) according as $i > j$ or $j > i$, respectively.

Therefore, the implementation of the action on face-pairs is not as simple as applying the action to each entry; we have to maintain the order's consistency.

⟨*how to act*⟩+≡
```
def act_face_pair(n,dp,facepair):
    (i,j) = facepair
    a_f = act_face
    (x,y) = (a_f (n,dp,i), a_f(n,dp,j))
    if x > y:
        return (x,y)
    else:
        return (y,x)
```

Finally, we represent a gluing as a ¿-sorted list of face-pairs. So we have to sort after applying the action on face-pairs.

⟨*how to act*⟩+≡
```
def act_gluing(n,dp,gluing):
    afp = act_face_pair
    unsorted = map((lambda fp: afp(n,dp,fp)), \
                                    gluing)
    return sorted(unsorted, reverse=True)
```

Having defined the action on gluings, we define the orbit. We only make sure that the list returned contains all elements from the orbit, and only elements from the orbit. We do not remove duplicates from this list.

⟨*how to orbit*⟩≡
```
def orbit_with_repeats(n,gluing):
    l = gluing
    nums = range(n)
    nums.reverse()
    ag = act_gluing
    symm_types = ["Rot","VRot","PRot","VPRot"]
    out = []
    for s in symm_types:
        for t in nums:
            out = [ag(n,(s,t),l)] + out
    return out
```

⟨*neckl.py*⟩+≡
⟨*how to orbit*⟩

Our tree will be a pretty standard binary tree on pairs of natural numbers. Its type's definition in Haskell would be

⟨*Haskell definition of the type of our tree*⟩≡
```
data PT = PTLeaf | PTBranch (Integer,Integer) PT PT
```

That is to say, a pair-tree is either a leaf, or it's a branch, with an associated pair (m, n) of integers and two associated pair-trees, a left child and right child. We again go with a poor-man's implementation of this datatype in Python. A pair-tree will be a tuple, either a singleton or a quadruple. If it is a singleton, its single entry will be the string `"PTLeaf"`. Otherwise, its initial entry will be the string `"PTBranch"`; its next entry will be a pair `(m,n)` of integers; and its final entries will be pair-trees.

The way we construct our tree follows from a more general construction of fixed-point-free involutions on arbitrary lists of even length. Suppose $fpf(l)$ were the set of fixed-point-free involutions on l. If l is empty, then this set is empty; we thus represent it by a leaf. Otherwise, l has at least two elements by evenness. We partition $fpf(l)$ according as an element does or does not flip the first two elements of l. Thus, $fpf(l) = (f\ f') \cdot fpf(l'') \cup X$, where f, f' are the first elements of l; l'' is l without these elements; and X is the subset of $fpf(l)$ sending f to something apart from f'.

At this point we end up with two options. Either we could use a non-binary tree, or we can introduce a sort of auxiliary stack parameter in fpf to "remember" which face-pairs we've foregone so far. Because our proof assistant Coq did not agree with non-binary trees, we choose the latter option.

So we end up with a slightly more complicated decomposition. We have some set m, and $fpf(l, m)$ is going to be the set of fixed-point-free involutions on $l \cup m$ *which sends the first element of* l *to some other element of* l. (Thus, our original definition of fpf would now be $fpf(l, \emptyset)$.) If l is empty, this is $\emptyset$. We represent this situation by a leaf. Otherwise, by l's evenness, it has two initial elements f, f'. Then we may decompose $fpf(l, m)$ into those involutions which flip f, f' and those which don't. We may write a fixed-point-free involution on $l \cup m$ which flips f, f' as the composition of $(f\ f')$ and a fixed-point-free involution of $l'' \cup m$, where l'' is l without f, f'. On the other hand, a fixed-point-free involution on $l \cup m$ which does not send f to f' but which does send f into l is a fixed-point-free involution on $l' \cup (\{f'\} \cup m)$ which sends f into some other element of l', where l' is l without f'. Thus we have a particular pair (f, f'), and two similar, simpler fpfs to consider. We represent this situation by a branch. (For convenience, we define an auxiliary "smush" function establishing union in the first part of the partition.)

⟨*how to make a prefix tree*⟩≡
```
  def smush (l,m):
      if m == []:
          return l
      else:
          return smush (([m[0]] + l), m[1:])

  def fpf(l,m):
      if l == []:
          return ("PTLeaf",)
      # else
      f = l[0]
      lp = l[1:]
      if lp == []:
          return ("PTLeaf",)
      fp = l[1]
      lpp = l[2:]
      return ("PTBranch", \
              (f,fp), \
              fpf(smush(lpp,m), []), \
              fpf([f] + lpp, [fp] + m))

  def original_tree(n):
      nums = range(2*n)
      nums.reverse()
      return fpf(nums, [])
```

⟨*neckl.py*⟩+≡
  ⟨*how to make a prefix tree*⟩

It's pretty simple to extract at least one involution from our tree if there is one. Just go left.

⟨*how to get an involution if there is one*⟩≡
```
  def get_left_inv(pt):
      if pt[0] == "PTLeaf":
          return []
      else:
          (str, p, lt, rt) = pt
          assert str == "PTBranch"
          return [p] + get_left_inv(lt)
```

⟨*neckl.py*⟩+≡
  ⟨*how to get an involution if there is one*⟩

It's also fairly straightforward to remove an involution from our tree. This, however, requires now that we spell out exactly what is the association between a path from root to leaf and fixed-point-free involution. By cycle notation it will suffice to describe how to extract from such a path a sequence of n face-pairs (in order to get a permutation in cycle notation). (The involutive and fixed-point-free properties must be proved from our original tree's properties, which proof we delay. TODO)

A path from root to leaf might be empty; in that case, the sequence is empty. Otherwise, the path takes some sequence of lefts and rights. We say that the sequence of a path is that list whose first element is the pair of the node at which the path first turns left, and whose tail is the associated sequence of the path from the root of the child tree to the given leaf.

The application of this for removing an involution is as follows. One has given an involution in the proper order, and a pair-tree again in proper order. (So if we were being thorough, we would need to define proper order and show the original tree is properly ordered, and show that removal preserves propriety.) If the tree is empty, you return the tree. Otherwise, compare the root with the head of the involution. If they differ, remove the whole involution from the right child. If, instead, they agree, then remove the rest of the involution from the left child. If the result is just a leaf, then we can get rid of this root entirely, so the result should be the right child. Otherwise, it's just the tree with the rest of the involution removed from the left child.

⟨*how to remove an involution*⟩≡

```
def remove_involution(l,pt):
    if l == []:
        return pt
    if pt[0] == "PTLeaf":
        return pt
    p,ps,(str,pp,lt,rt) = l[0],l[1:],pt
    assert str == "PTBranch"
    if p != pp:
        return ("PTBranch", pp, lt, \
                remove_involution(l,rt))
    ltp = remove_involution(ps,lt)
    if ltp[0] == "PTLeaf":
        return rt
    else:
        return ("PTBranch", pp, ltp, rt)
```

⟨*neckl.py*⟩+≡

⟨*how to remove an involution*⟩

This takes time proportional to at most the number of face-pairs. With a more sophisticated structure, we could possibly get it down lower. At any rate, this is much better than time proportional to the number of *gluings*.

In conclusion for this section, it's now completely straightforward to implement our original plan as described above: pop top, remove orbit, repeat.

⟨*how to get orbit representatives*⟩≡
```
def orbit_reps(n):
    pt = original_tree(n)
    reps = []
    while pt[0] != "PTLeaf":
        l = get_left_inv(pt)
        reps = [l] + reps
        lorb = orbit_with_repeats(n,l)
        for lp in lorb:
             pt = remove_involution(lp,pt)
    return reps
```

⟨*neckl.py*⟩+≡
⟨*how to get orbit representatives*⟩

Vetting the manifolds We can now generate all the manifolds of interest to us. So now our task is to determine among these which can embed nonelementarily into a one-cusped hyperbolic 3-manifold.

We can first implement some basic sanity checks. All necklace gluings are orientable. Likewise, they are all connected. (If we were testing arbitrary MOM gluings, then we would add a connectedness test.) Our other requirement, though, is that the links of all the vertices must be tori. This is known in the literature as a "(generalized) link complement," i.e. the complement of a link in some closed 3-manifold.

⟨*link complement*⟩≡
```
def link_complement(mfld):
    vs = mfld.vertices()
    for v in vs:
        if not (v.link() == v.TORUS):
            return False
    else:
        return True
```

⟨*neckl.py*⟩+≡
⟨*link complement*⟩

Now comes the real work. We now find necessary conditions for a manifold to embed nonelementarily into a one-cusped hyperbolic 3-manifold.

Suppose now that M embeds nonelementarily into a one-cusped hyperbolic 3-manifold N. If M has an essential sphere S, let $M = M_0 \# M_1$ be the associated connect-sum decomposition of M. Because S bounds a ball in N, one of M_0, M_1 is an S^3-link-complement. But M's connect-sum decomposition is not empty, since S^3 doesn't embed nonelementarily into N. So M has a non-empty connect-sum decomposition with at most one summand which is not an S^3-link-complement. So it will behoove us to get statistics on the lengths of the connect-sum decompositions of our given manifolds. Regina only implements connect-sum decompositions for closed 3-manifolds, so we roll our own routine here.

⟨*connect-sum decomposition*⟩≡
```
def slow_connect_sum(regina_mfld):
    M = NTriangulation(regina_mfld)
    if not M.isConnected():
        raise Exception("Not connected")
    if not M.isOrientable():
        raise Exception("Not orientable")
    if not link_complement(M):
        raise Exception("Not a generalized link complement")
    nsl = NNormalSurfaceList.enumerate
    l = nsl(M, NS_STANDARD, \
            NS_FUNDAMENTAL)
    n = l.getNumberOfSurfaces()
    for i in range(0,n):
        s = l.getSurface(i)
        x = s.eulerChar()
```

⟨*neckl.py*⟩+≡
```
⟨connect-sum decomposition⟩
```

For completeness we deal with the possibility of projective planes. Of course, if M has an embedded projective plane, then it cannot embed into N.

⟨*connect-sum decomposition*⟩+≡
```
        if x != 2:
            if x != 1:
                continue
            if s.isOrientable():
                continue
```

At this point we know $s = P^2$. This should be impossible, so we throw an exception. This will make the routine less useful for other folks, but it ensures that we've exhausted the possibilities for reducibility.

⟨*connect-sum decomposition*⟩+≡
```
            raise Exception("P^2!")
```

At this point we know s is a sphere. Cut M along s into m. Then m has two sphere boundary components. We cap them off by idealizing m.

We note here that Python includes indentation as a critical part of its syntax. It's somewhat hard to tell in this PDF that the following code is indented outside the previous large `if` statement but inside the overall `for` loop. The statements inside the `if` are at indentation level at least 3; the following is at indentation level 2.

⟨*connect-sum decomposition*⟩+≡

```
m = s.cutAlong()
m.finiteToIdeal()
m.intelligentSimplify()
```

If m is connected, then s was non-separating, and established the connect-sum $M = (S^2 \times S^1)\#m$. This is impossible for non-elementary embeddings, so again we raise an exception.

⟨*connect-sum decomposition*⟩+≡

```
if m.isConnected():
    raise Exception("Non-separating S^2!")
```

Now, suppose instead that s separates M. Then s is essential precisely when neither component of M cut along s is a ball. We've idealized the cut, so this is equivalent to neither component of m being S^3.

⟨*connect-sum decomposition*⟩+≡

```
m.splitIntoComponents()
m0 = m.getFirstTreeChild()
m1 = m0.getNextTreeSibling()
if not (m0.isThreeSphere() or \
        m1.isThreeSphere()):
    return slow_connect_sum(m0) + \
           slow_connect_sum(m1)
```

The following is at indentation level 1, outside the `for` loop. This code is only run if none of the above `raise` or `return` statements are encountered. In that case, there were no interesting surfaces. But if there is no essential sphere, then M is irreducible. So we just need to determine whether or not it's S^3. (No necklace manifold is S^3; they all have boundary.)

⟨*connect-sum decomposition*⟩+≡

```
    if M.isThreeSphere():
        return []
    else:
        return [M]

def vet_reds(sane_mflds):
    oks = []
    bads = []
    for i in range(len(sane_mflds)):
        mfld = sane_mflds[i]
        try:
            csd = slow_connect_sum(mfld)
            oks = [((i,mfld),csd)] + oks
        except:
            bads = [(i,mfld)] + bads
    return (oks,bads)
```

There are only 3 sane 4-bead necklace gluings and 13 sane 5-bead necklace gluings, and they are all irreducible.

As it turns out, 3 sane 6-bead necklace gluings have an embedded P^2; these cannot embed nonelementarily. Then there are 16 6-bead necklace gluings which have a nontrivial connect-sum decomposition and don't have a P^2. But, happily, they are all of the form $(D^2 \times S^1)\#(D^2 \times S^1)$. This does not embed nonelementarily into hyperbolic 3-manifolds. The remaining 88 sane 6-bead necklace gluings are irreducible.

Suppose now that we have an irreducible manifold M which embeds nonelementarily into N. If M is not hyperbolic, then it has an essential torus. (One can, in fact, show something more, about which more later should it prove necessary.) (And, of course, conversely, if it has an essential torus, then it is not hyperbolic.) So let's look for essential tori and cut along them until we can do so no more.

N. B. By "essential torus," I mean a properly embedded incompressible torus which is not boundary parallel.

N. B. Cutting along essential tori until there are no more (which is what we are doing) is not how the JSJ decomposition works. The JSJ decomposition is more subtle than that. In particular, Seifert-fibered manifold is not necessarily atoroidal, but its JSJ decomposition is just itself.

Neil Hoffman and I have found a way to speed up the $T^2 \times I$ test from my thesis using Berge and Dave's work on fillings of links in solid tori; I use that here.

I also add some basic diagnostic stuff on the side to see what's going on while the procedure.

⟨*cut along essential tori*⟩≡
```
from new_unhyp import new_isT2xI as isT2xI
def cut_essential_tori(regina_mfld, loc):
    print loc
    M = NTriangulation(regina_mfld)
    M.finiteToIdeal()
    M.intelligentSimplify()
    nsl = NNormalSurfaceList.enumerate
    l = nsl(M, NS_STANDARD, \
                          NS_FUNDAMENTAL)
    n = l.getNumberOfSurfaces()
    print "There are " + str(n) + " surfaces"
    for i in range(0,n):
        print "Working on surface " + str(i)
        s = l.getSurface(i)
        x = s.eulerChar()
        if x != 0:
            continue
        if s.hasRealBoundary():
            continue
```

⟨*neckl.py*⟩+≡
⟨*cut along essential tori*⟩

Again, for completeness' sake we make an allowance for Klein bottles in the form of an exception.

⟨*cut along essential tori*⟩+≡
```
        if not s.isOrientable():
            raise Exception("K^2!")
```

Having located a fundamental normal torus, we should test whether or not it is essential. This boils down to whether or not it (co)bounds either a solid torus (implying it is compressible) or a $T^2 \times S^1$ (implying it is boundary-parallel).

Of course, we should first ensure that s is not a vertex link; these are never essential.

⟨*cut along essential tori*⟩+≡
```
if s.isVertexLinking():
    continue
```

Of next concern to us, as above, is whether or not it separates. It ought to! So if it doesn't, we raise an exception.

⟨*cut along essential tori*⟩+≡
```
m = s.cutAlong()
if m.isConnected():
    raise Exception("Non-separating T^2!")
```

Having found a separating torus, we can now test whether or not it is essential, by testing whether the components of m are either $D^2 \times S^1$ or $T^2 \times I$.

⟨*cut along essential tori*⟩+≡
```
m.finiteToIdeal()
m.intelligentSimplify()
m.splitIntoComponents()
m0 = m.getFirstTreeChild()
m1 = m0.getNextTreeSibling()
print "Left solid torus?"
if m0.isSolidTorus():
    continue
print "Right solid torus?"
if m1.isSolidTorus():
    continue
print "Left T2 x I?"
if isT2xI(m0):
    continue
print "Right T2 x I?"
if isT2xI(m1):
    continue
```

If they are not, then the torus is essential. So we look for other tori in the components. I'm not sure how to locate the boundary tori along which `m0` and `m1` glue up to M, let alone what the gluing map is. It would be very nice to remember such information in a tree-like structure, which we could implement at this juncture. So instead of just shoving the components in a list, we provide scaffolding for such an improvement by returning a pair if there's an essential torus.

⟨*cut along essential tori*⟩+≡
```
return (cut_essential_tori(m0, loc+"L"), \
                cut_essential_tori(m1, loc+"R"))
```

Otherwise, if there is no essential torus, then we've ended our cutting, and just return M itself. Again, for scaffolding purposes, we put it in a uniple.

⟨*cut along essential tori*⟩+≡
```
    else:
        return (M,)
```

Now we make a function which culls through a list or irreducibles, throwing out anything with a "bad" surface (e.g. like a K^2).

⟨*vetting manifolds*⟩≡
```
def vet_tor(idx_irreds):
    cut_mflds = []
    bad_mflds = []
    for p in idx_irreds:
        (i,m) = p
        print str(i)
        try:
             cut_up = cut_essential_tori(m,".")
             cut_mflds = [(i,cut_up)] + cut_mflds
        except Exception as uh_oh:
            s = ["P^2!", "K^2!"]
            s = s + ["Non-separating S^2!"]
            s = s + ["Non-separating T^2!"]
            if uh_oh.message in s:
                bad_mflds = [(i,m,uh_oh.message)] + bad_mflds
            else:
                raise Exception(uh_oh.message)
    return (cut_mflds, bad_mflds)
```

⟨*neckl.py*⟩+≡
```
⟨vetting manifolds⟩
```

By inspection, all but one 4-bead gluing are atoroidal. Conveniently, the same is true of 5-bead gluings. We explain the toroidal ones after dealing with the 6-beads.

All but 20 6-bead gluings are atoroidal, and among these, 6 have an embedded K^2. Each of the remaining 14 splits along a torus into two atoroidal pieces. The fundamental groups of 10 of these pieces manifestly admit only elementary embeddings into $PSL_2\mathbf{C}$. These manifolds are in Table 1. We give them as both their necklace gluing and their Regina isomorphism signature, and their "children" just as isomorphism signatures. The remaining 4 manifolds have only three isomorphism types, as given in Table 2.

To deal with the remaining three 6-bead manifolds (four gluings), we examine their fundamental groups, and more importantly the fundamental groups of their atoroidal pieces, as follows.

IsoSig	Gluing	π_1 Presentation
gLLAQbcdefffhbbhabb	(11 10)(9 1)(8 5)(7 0)(6 2)(4 3)	$\langle a, b \mid a^{-2}b^2a^2b^{-2}\rangle$
gLLAQbeddfffaabgabg	(11 10)(9 4)(8 1)(7 3)(6 2)(5 0)	$\langle a, b \mid a^{-2}b^{-2}a^2b^2\rangle$
gLLAQbeddfffhhbghgg	(11 10)(9 4)(8 3)(7 0)(6 2)(5 1)	$\langle a, b \mid a^2b^{-2}a^{-2}b^2\rangle$
gLLMQbcdfeffagbaabg	(11 10)(9 1)(8 4)(7 0)(6 2)(5 3) (11 10)(9 5)(8 0)(7 2)(6 3)(4 1)	$\langle a, b \mid a^{-2}b^{-2}a^2b^2\rangle$
gLLPQccdfeffabghhbg	(11 10)(9 5)(8 1)(7 3)(6 2)(4 0)	$\langle a, b \mid a^{-2}b^{-2}a^2b^2\rangle$
gLLPQcdeffefafqqaqo	(11 10)(9 1)(8 0)(7 4)(6 3)(5 2) (11 10)(9 1)(8 0)(7 5)(6 3)(4 2)	$\langle a, b \mid a^2b^2a^{-2}b^{-2}\rangle$
gLLPQcdeffefqfaqaqo	(11 4)(10 1)(9 0)(8 5)(7 3)(6 2) (11 4)(10 2)(9 1)(8 0)(7 5)(6 3)	$\langle a, b \mid a^{-2}b^3a^2b^{-3}\rangle$

Table 1: The obviously elementary 6-bead irreducible toroidal necklace gluings.

IsoSig	Gluing
gLLMQbeeffefxxxxxax	(11 10)(9 4)(8 5)(7 2)(6 1)(3 0)
gLLPQcdfeeffafaaaaf	(11 10)(9 1)(8 4)(7 5)(6 2)(3 0) (11 10)(9 7)(8 6)(5 2)(4 1)(3 0)
gLLMQcecefffhabaahb	(11 10)(9 2)(8 4)(7 5)(6 1)(3 0)

Table 2: The non-trivial 6-bead irreducible toroidal necklace gluings.

Parent IsoSig	Child π_1 Presentations
gLLMQbeeffefxxxxxax	$\langle a, b \mid a^{-1}b^{-1}a^{-1}b^3\rangle$, $\langle a, b, c \mid a^{-1}bab^{-1},\ b^1c^{-1}bc\rangle$
gLLPQcdfeeffafaaaaf	$\langle a, b, c \mid ab^{-1}a^{-1}b, ab^{-1}ca^{-1}bc^{-1}\rangle$ $\langle a, b \mid a^{-2}b^{-1}a^2b\rangle$
gLLMQcecefffhabaahb	$\langle a, b, c \mid b^{-1}cbc^{-1}, a^{-1}bc^{-1}ab^{-1}c\rangle$ $\langle a, b \mid a^{-3}b^3\rangle$

Except for the first group presentation, these presentations show that these groups do not admit discrete, torsion-free, nonabelian representations into $PSL_2\mathbf{C}$, because hyperbolic and parabolic elements of $PSL_2\mathbf{C}$ commute if and only if they share an axis, and a power of such an element has the same axis as the element.

For the first group, $G_0 = \langle a, b \mid a^{-1}b^{-1}a^{-1}b^3\rangle$, we can introduce $c = a^{-1}b^{-1}$ to get $G_0 = \langle a, b, c \mid abc, c^2b^4\rangle$. But abc is a defining relator for a, so we can remove it and get $G_0 = \langle b, c \mid c^2b^4\rangle$, which now plainly doesn't admit such a representation into $PSL_2\mathbf{C}$.

So, for each of the three remaining 6-bead 3-manifolds M, $\pi_1(M)$ is a pushout of groups H_0, H_1 along a common subgroup $H = \mathbf{Z}^2$, and the only discrete torsion-free representations of H_0, H_1 into $PSL_2\mathbf{C}$ are abelian. But then for any representation $\rho : \pi_1(M) \to PSL_2\mathbf{C}$, $\rho(H_0), \rho(H_1)$ are abelian subgroups of $PSL_2\mathbf{C}$ with a common abelian subgroup. Thus all $\rho(\pi_1(M))$ is abelian. Therefore, none of the remaining 3-manifolds M embed nonelementarily in a hyperbolic N.

The unique irreducible toroidal 4-bead manifold, gluing (7 2)(6 0)(5 3)(4 1) has isomorphism signature `eLPkbdcddhbbbg` and fundamental group presentation $\langle a, b \mid a^2b^{-2}a^{-2}b^2\rangle$. So it does not embed nonelementarily.

Finally, the unique irreducible toroidal 5-bead manifold, gluing (9 8)(7 3)(6 4)(5 1)(2 0), has isomorphism signature `fLLQcbeddeeahbgah`. Its atoroidal pieces have fundamental groups $\langle a, b, c \mid aca^{-1}c^{-1}, bc^{-1}b^{-1}c\rangle$ and $\langle a, b \mid a^3b^{-2}\rangle$. By the pushout argument above, this manifold doesn't embed nonelementarily in a hyperbolic N.

Now, as for the atoroidal remainder, there will be hyperbolic and Seifert-fibered manifolds. We can distinguish the Seifert-fibered manifolds by finding essential annuli in them, in fact by finding non-separating annuli in them. Hyperbolic manifolds have no such surface. Those manifolds which do not have a non-separating annulus will be hyperbolic. We identify these with entries from the SnapPea census.

We only look for vertex normal annuli, as you may check in the `findNonSeparatingAnnulus` routine. If anything, this will only accidentally put some SFS in with the hyperbolics—not the other way round.

⟨*hyperbolic and Seifert-fibered*⟩≡
```
from new_unhyp import findNonSeparatingAnnulus
from snappy import *
def hyp_or_SFS(atoroidals):
    hyps = []
    sfss = []
    for p in atoroidals:
        (i,(m,)) = p
        print "Working on " + str(i)
        if m.hasStrictAngleStructure():
            hyps = [p] + hyps
            continue
        m.idealToFinite()
        m.intelligentSimplify()
        a = findNonSeparatingAnnulus(m)
        if a == None:
            hyps = [p] + hyps
            continue
        else:
            sfss = [(a,p)] + sfss
            continue
    return (hyps, sfss)
```

⟨*neckl.py*⟩+≡
⟨*hyperbolic and Seifert-fibered*⟩

This leaves us with a list of 17 potential hyperbolic manifolds, which Regina verifies as hyperbolic (it can find strict angle structures on them), and 51 Seifert-fibered spaces with their associated non-separating annuli.

We would like to identify the 17 hyperbolic 6-bead manifolds, which, by the above, are the only 6-bead necklace gluings which embed nonelementarily in a hyperbolic N. This is relatively straightforward using `SnapPy`'s `identify` procedure. You may check that there are 6 `m129`s, 2 `s776`s, 1 `s780`, 1 `s443`, 1 `s647`, 2 `s785`s, 1 `s782`, 1 `m292`, and 1 `s596`, and 1 `s774`.

Bead number	Necklace embeddables
4	m129
5	m125, m203, m295
6	m129, m292, s443, s596, s647, s774, s776, s780, s782, s785

Finally, for the lower bead numbers. There are three hyperbolic 5-bead necklace gluings, which, again, are the only 5-bead necklace gluings which embed nonelementarily in a hyperbolic N. They are m295, m203, and m125. And there is a unique 4-bead hyperbolic necklace gluing, the unique 4-bead necklace gluing which embeds nonelementarily in a hyperbolic N. It is m129, the Whitehead link.

In summary, the following are the unique d-bead necklace gluings which embed nonelementarily into a hyperbolic N; they are all hyperbolic census manifolds.