Week 4

Black Box Testing - II

In this chapter, we shall discuss the following types of black box testing:

- Boundary value testing
- Combinatorial Testing
- Pairwise testing.

Last week we discussed Random, Equivalence and Category partitioning testing techniques. Let us continue from there and discuss more types.

Boundary Value Testing (Boundary Value Analysis)

Off-by-one mistakes are a common cause for bugs in software systems. As developers, we have all made mistakes such as using a "greater than" operator (>) where it had to be a "greater than or equal to" operator (>=). Interestingly, programs with such a bug tend to work well for most of the provided inputs. They fail, however, when the input is "near the boundary of condition".

Boundaries in between classes/partitions

Whenever we devised partitions/classes, these classes have *"close boundaries"* with the other classes. In other words, if we keep performing small changes to an input that belongs to some partition (e.g., by adding +1 to it), at some point this input will belong to another class. The precise point where the input changes from one class to another is what we call a boundary. And this is precisely what boundary testing is about: to make the program behave correctly when inputs are near a boundary. We can find such boundaries by finding a pair of consecutive input values [p_1, p_2], where p_1 belongs to partition A, and p_2 belongs to partition B. In other words, the boundary itself is where our program changes from our class to the other. As testers, we should make sure that everything works smoothly (i.e., the program still behaves correctly) near these values.

Example 8:

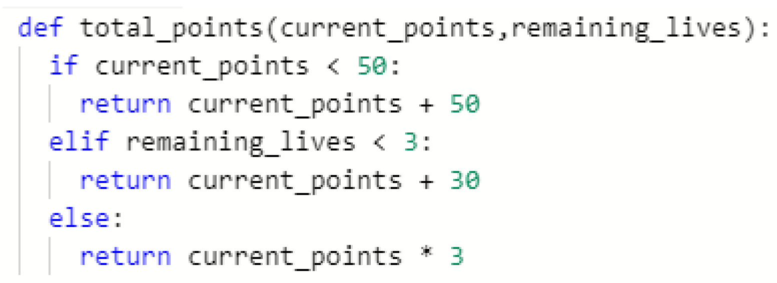
Requirement: Calculating the amount of points of the player

Given the score of a player and the number of remaining lives of the player, the program does the following:

- If the player's score is below 50, then it always adds 50 points on top of the current points.
- If the player's score is greater than or equals to 50, then:

- if the number of remaining lives is greater than or equal to 3, it triples the score of the player.
- otherwise, it adds 30 points on top of the current points.

If written in Python, it will look like as under:



```python
def total_points(current_points,remaining_lives):
    if current_points < 50:
        return current_points + 50
    elif remaining_lives < 3:
        return current_points + 30
    else:
        return current_points * 3
```

Explanation:

When devising the partitions to test this method, we come up with the following partitions:

- Score < 50
- Score >= 50 and remaining life < 3
- Score >= 50 and remaining life >= 3

Boundary 1: When the score is strictly smaller than 50, it belongs to partition 1. If the score is greater than or equal to 50, it belongs to partitions 2 and 3. Therefore, we observe the following boundary: when the score changes from 49 to 50, the partition it belongs to also changes (let us call this test B1).

Boundary 2: Given a score that is greater than or equal to 50, we observe that if the number of remaining lives is smaller than 3, it belongs to partition 2; otherwise, it belongs to partition 3. Thus, we just identified another boundary there (let us call this test B2). We can visualize these partitions with their boundaries in a diagram.

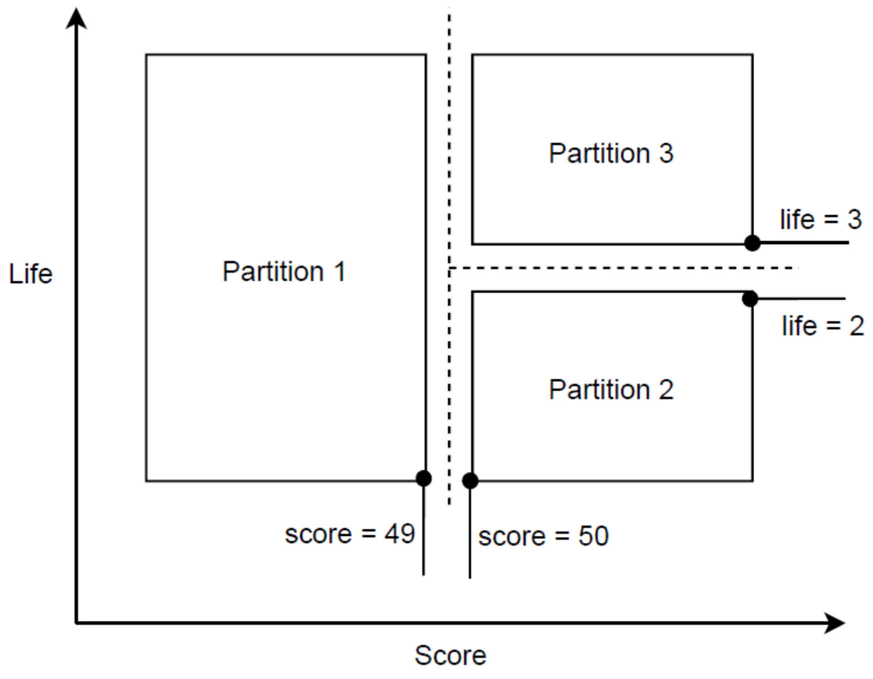


Figure 1: Boundaries

To sum up: you should devise tests for the inputs at the boundaries of your classes.

For B1:

B1.1 = input= {score=49, remaining lives=5}, output= {99}
B1.2 = input= {score=50, remaining lives=5}, output= {150}

For B2:

B2.1 = input= {score 500, remaining lives=3}, output= {1500}
B2.2 = input= {score 500, remaining lives=2}, output= {530}

Analysing Boundary Conditions (On & Off Points)

Errors where the system behaves incorrectly for values on and close to the boundary are very easily made. In practice, think of how many times the bug was in the boolean condition of your *if* or *for* condition, and the fix was basically replacing a ≥ by a >. When we have a properly specific condition, e.g., x > 100, we can analyse the boundaries of this condition.

First, we need to go over some terminology:

On-point: the value that is on the boundary. This is the value we see in the condition.
Off-point: the value closest to the boundary that flips the conditions. So, if the on-point makes the condition true, the off point makes it false and vice versa. Note: when dealing with equalities or non-equalities (e.g. x = 6 or x ≠ 6), there are two off-points; one in each direction.

In-points are all the values that make the condition true.
Out-points are all the values that make the condition false.

Note that, depending on the condition, an on-point can be either an in- or an out-point.

Example 9:

Requirement:

Suppose we have a program that adds shipping costs when the total price is below 100.

The condition used in the program is x < 100.

- The on point is 100, as that is the value in the condition.
- The on-point makes the condition false, so the off-point should be the closest number that makes the condition true. This will be 99, 99 < 100 is true.
- The in-points are all the values smaller than or equal to 99. For example, 37, 42, 56.
- The out-points are all values larger than or equal to 100. For example, 325, 1254, 101.

.
We show all these points in the diagram below.

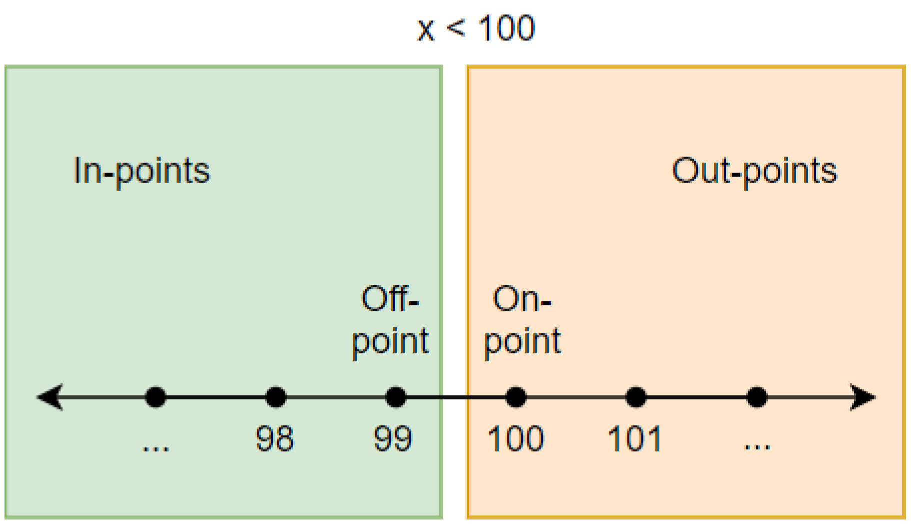


Figure 2: In out points if x < 100

Now, let us compare it to the next condition x <= 100 (note how similar they are; the only difference is that, in this one, we use smaller than or equal to):

- The on point is still 100: this is the point in the condition
- Now the condition is true for the on-point. So, the off-point should make the condition false; the off-point is 101.

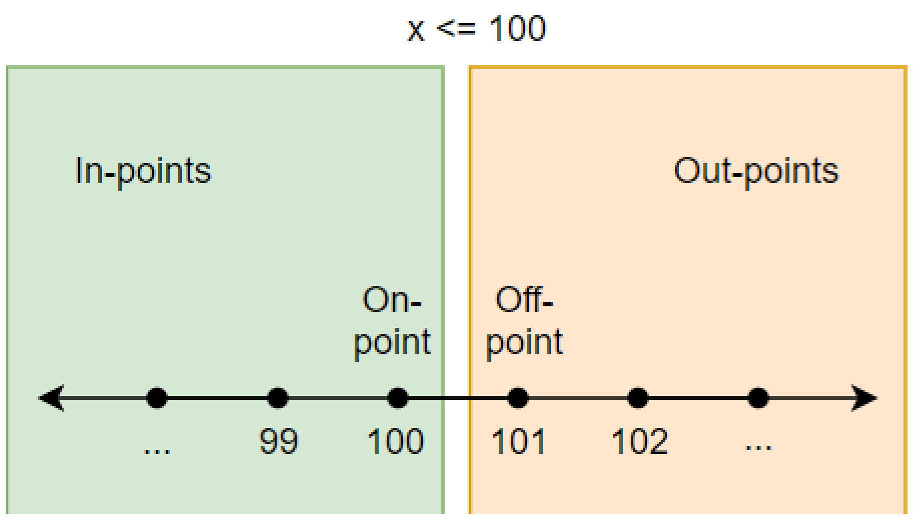


Figure 3: In and Out points

Note that, in the diagram, the on-point is part of the in-points, and the off-point is part of the out-points.

Implicit Boundaries

Let's revisit the example 5, where the goal was to return the number of bars needed in order to build some boxes of chocolates:

A developer developed the program in Python Programming Language as:

```python
def calculate(small, big, total):
    max_boxes = total/5
    big_boxes_we_can_use = min(max_boxes,big)
    total -= big_boxes_we_can_use * 5

    if small <= total:
        return -1
    else:
        return total
```

However, another developer tried (2,3,17) as an input and the program crashed. After some debugging, they noticed that the if statement should have been if(small < total) instead of if(small <= total). This smells like a bug that could have been caught via boundary testing. The problem is that this boundary is just less explicit from the requirements.

Note that the test (2,3,17) belongs to the need small + big bars partition. In this case, the program will make use of all the big bars (there are 3 available) and then all the small bars available (there are 2 available). Note that the buggy program would work if we had 3 available small bars (having (3, 3, 17) as input). This is a boundary.

Boundaries also happen when we are going from “one partition” to another. In these cases, what we should do is to devise test cases for a sequence of inputs that move from one partition to another.

For example, let us focus on the bug caused by the (2,3,17) input.

- (1,3,17) should return not possible (1 small bar is not enough). This test case belongs to the not enough bars partition.
- (2,3,17) should return 2. This test case belongs to the need for small + big bars partition.

There is a boundary between the (1,3,17) and the (2,3,17). We should make sure the software still behaves correctly in these cases.

Let’s explore another boundary. Let us focus on the only big bars partition. We should find inputs that transition from this partition to another one:

- (10, 1, 10) returns 5. This input belongs to the need small + big bars partition.
- (10, 2, 10) returns 0. This input belongs to the need only big bars partition.

One more? Let us focus on the only small bars partition:

- (3, 2, 3) returns 3. We need only small bars here, and therefore, this input belongs to the only small bars partition.
- (2, 2, 3) returns -1. We cannot make the boxes. This input belongs to the Not enough bars partition.

A partition might make boundaries with other partitions. See:

- (4, 2, 4) returns 4. We need only small bars here, and therefore, this input belongs to the only small bars partition.
- (4, 2, 5) returns 0. We need only big bars here, and therefore, this input belongs to the only big bars partition.

Your lesson is: explore the boundaries in between your partitions!

Combinatorial Testing

Combinatorial testing is a method that can reduce the cost and improve the test effectiveness significantly for many applications. The key insight underlying this form of testing is that not every parameter contributes to every failure, and empirical data suggest that nearly all software failures are caused by interactions between relatively few parameters.

Pairwise Testing (All Pairs Testing)

In pairwise testing, we design test cases to execute all possible discrete combinations of each pair of input parameters. Pairwise testing requires that for a given numbers of input parameters to the system each possible combination of values for any pair of parameters be covered by at least one test case. As mentioned earlier, the output of a software application depends on many factors e.g. input parameters, state variables and environmental configuration. Techniques like boundary value analysis and equivalence partitioning can be useful to identify the possible values for individual factors as discussed earlier. But it is impractical to test all possible combinations of values for all those factors so instead a subset of combinations is generated to satisfy all factors. Tests are designed such that for each pair of input parameters to a system, there are all possible discrete combinations of those parameters. The test suite covers all combinations; therefore it is not exhaustive yet very effective in finding bugs.

Category partitioning, discussed earlier, works well when intuitive constraints reduce the number of combinations to a small amount of test cases. So, without any constraints the number of combinations is unmanageable. Pairwise testing (instead of exhaustive approach) generates combinations that efficiently cover all partitions with a smaller number of test cases. Let us see an example.

Example 10: Display control of the system

For a display system have following options

Display mode: graphics, text only and limited bandwidth.
Supported languages: English, French, Mandarin and Arabic.
Fonts: minimal, standard, and document loaded.
Colours: monochrome, colormap, 16-bit, True colour
Screen-Size: Laptop, Full Size, Handheld.

With category partitioning, we can have following number of test cases

3*4*3*4*3 = 432.

While applying pairwise testing for this example we come up with 17 tests only...How?? We take any of the options as a standard (say language in this case) and make pairs with other combinations.

1. English,Monochrome,Full-graphics,Minimal,Hand-held
2. English,Color-map,Text-only,Standard,Full-size
3. English,16-bit, Limited-bandwidth, -, Full-size
4. English,True-color,Text-only,Document-loaded,Laptop

5. French,Monochrome,Limited-bandwidth,Standard,Laptop
6. French,Color-map,Full-graphics,Document-loaded,Full-size
7. French,16-bit, Text-only,Minimal, -
8. French, True-color, -, -, Hand-held
9. Arabic,Monochrome, -, Document-loaded, Full-size
10. Arabic, Color-map, Limited-bandwidth, Minimal, Hand-held
11. Arabic, 16-bit, Full-graphics, Standard, Laptop
12. Arabic, True-color, Text-only, -, Hand-held
13. Mandarin, -, -, Monochrome, Text-only
14. Mandarin, Color-map, -, Minimal, Laptop
15. Mandarin, 16-bit, Limited-bandwidth, Document-loaded, Hand-held
16. Mandarin, True-color, Full-graphics, Minimal, Full-size
17. Mandarin, True-color, Limited-bandwidth, Standard, Hand-held

Example 11: Car Ordering Application:

The car ordering application allows for Buying and Selling cars. It should support trading in Melbourne and Sydney. The application should have registration numbers, may be valid or invalid. It should allow the trade of following cars: BMW, Audi, and Mercedes. Two types of booking can be done: E-booking and In Store. Orders can be placed only during trading hours.

1. Let's list down the categories and respective choices we have
   a. Order Types
      i. Buy
      ii. Sell
   b. Locations
      i. Melbourne
      ii. Sydney
   c. Car Models
      i. BMW
      ii. Audi
      iii. Mercedes
   d. Registration Numbers
      i. Valid (let us say ABC1500)
      ii. Invalid
   e. Order Types
      i. E-Booking (I call it online for time being)
      ii. In-Store
   f. Order Times
      i. Working hours (we can also call it trading hours)
      ii. Non-working hours

2. Let’s arrange it in a table format

Order Category	***Locations***	***Models***	***Registration***	***Order Type***	***Order Time***
Buy	Melbourne	BMW	Valid	Online	Working Hours
Sell	Sydney	Audi	InValid	In Store	Non-Working Hours
		Mercedes			

3. Arrange choices to create a test suite

Let’s start filling in the table column by column. Initially, the table should look something like this. The three values of Model (having the highest number of values) should be written two times each (two is the number of values of next highest variable i.e. Order category).

Model	***Order Category***	***Location***	***Registration***	***Order Type***	***Order Time***
BMW					
BMW					
Audi					
Audi					
Mercedes					
Mercedes					

Next choice is order category so fill it in.

Model	***Order Category***	***Location***	***Registration***	***Order Type***	***Order Time***
BMW	Buy				
BMW	Sell				
Audi	Buy				
Audi	Sell				
Mercedes	Buy				
Mercedes	Sell				

Model	***Order Category***	***Location***	***Registration***	***Order Type***	***Order Time***
BMW	Buy	Melbourne			
BMW	Sell	Sydney			
Audi	Buy	Melbourne			
Audi	Sell	Sydney			
Mercedes	Buy	Melbourne			
Mercedes	Sell	Sydney			

Hang on we have *buy and Melbourne* but no *buy and Sydney* so let's swap the choice say with Audi

Model	***Order Category***	***Location***	***Registration***	***Order Type***	***Order Time***
BMW	Buy	Melbourne			
BMW	Sell	Sydney			
Audi	Buy	Sydney			
Audi	Sell	Melbourne			
Mercedes	Buy	Melbourne			
Mercedes	Sell	Sydney			

This looks better. Let's repeat the same steps for remaining columns.

Model	***Order Category***	***Location***	***Registration***	***Order Type***	***Order Time***
BMW	Buy	Melbourne	Valid		
BMW	Sell	Sydney	Invalid		
Audi	Buy	Sydney	Valid		
Audi	Sell	Melbourne	Invalid		
Mercedes	Buy	Melbourne	Valid		
Mercedes	Sell	Sydney	Invalid		

When columns 3 and 4 are compared, each value in column 3 has both the values of column 4. But when you compare the 2nd and 4th column, we have Buy and Valid & Sell and Invalid that is Buy does not have 'Invalid' and Sell does not have 'Valid'. Hence, we need to interchange the last set of values in the 4th column.

Model	***Order Category***	***Location***	***Registration***	***Order Type***	***Order Time***
BMW	Buy	Melbourne	Valid		
BMW	Sell	Sydney	Invalid		
Audi	Buy	Sydney	Valid		
Audi	Sell	Melbourne	Invalid		
Mercedes	Buy	Melbourne	Invalid		
Mercedes	Sell	Sydney	valid		

Let's fill the remaining columns

Model	***Order Category***	***Location***	***Registration***	***Order Type***	***Order Time***
BMW	Buy	Melbourne	Valid	Instore	Working Hours
BMW	Sell	Sydney	Invalid	Online	Non-Working Hours

Audi	Buy	Sydney	Valid	Online	Working Hours
Audi	Sell	Melbourne	Invalid	Instore	Non-Working Hours
Mercedes	Buy	Melbourne	Invalid	Online	Working Hours
Mercedes	Sell	Sydney	valid	Instore	Non-Working Hours

Column 6 (Order time) is problematic. We are missing Buy/Non-working hours and Sell/Working hours. We can't fit our missing pairs by swapping around values as we already swapped all the rows if we swap now, we may miss other possible pairs which are already sorted. So, we add two more test cases that contain these pairs. Hence, the blank rows!

Model	***Order Category***	***Location***	***Registration***	***Order Type***	***Order Time***
BMW	Buy	Melbourne	Valid	Instore	Working Hours
BMW	Sell	Sydney	Invalid	Online	Non-Working Hours
					Non-Working Hours
Audi	Buy	Sydney	Valid	Online	Working Hours
Audi	Sell	Melbourne	Invalid	Instore	Non-Working Hours
					Working Hours
Mercedes	Buy	Melbourne	Invalid	Online	Working Hours
Mercedes	Sell	Sydney	valid	Instore	Non-Working Hours

Now we will fill in the empty cells as we desire because the other variable values are purely arbitrary

Model	***Order Category***	***Location***	***Registration***	***Order Type***	***Order Time***
BMW	Buy	Melbourne	Valid	Instore	Working Hours
BMW	Sell	Sydney	Invalid	Online	Non-Working Hours
BMW	Buy	Melbourne	Valid	Instore	Non-Working Hours
Audi	Buy	Sydney	Valid	Online	Working Hours

Audi	Sell	Melbourne	Invalid	Instore	Non-Working Hours
Audi	sell	Sydney	Invalid	Online	Working Hours
Mercedes	Buy	Melbourne	Invalid	Online	Working Hours
Mercedes	Sell	Sydney	valid	Instore	Non-Working Hours

So, the 8 tests are enough to test all categories and choices using pairwise testing…. Hurrah.

Jenny Tool

The public domain `jenny` combinatorial testing tool [8] can generate test frames for us - or at least, *covering arrays* which we can turn into test frames:

```
xubuntu@xubuntu-vb: ./jenny -n2 3 2 2
1a                                2b                                3b
1b                                2a                                3a
1c                                2a                                3b
1b                                2b                                3b
1a                                2b                                3a
1c                                2b                                3a
1a 2a 3a
```

In the category-choice framework, the digits represent categories and the letters choices, so "1a 2b 3b" represents "first category, first choice", "second category, second choice", and "third category, second choice".

You can extend the concept of pairwise combinations to t-wise combinations, for any value of t, with a corresponding increase in fault-detection effectiveness as shown in the graph above.

`jenny` (as with many other tools) can generate t-wise covering arrays for doing t-wise combinatorial testing, but they are *not* guaranteed to be as small as possible. However, jenny performs surprisingly well against much of the competition, including commercial tools, according to a comparison at pairwise.org (which lists a wide variety of other tools) [9].

Choosing black-box techniques

So, we've presented a variety of black box testing techniques here. How can we choose appropriate techniques for testing a software artifact we'd like to test?

The first thing to note is that the test case selection techniques presented here are *not* drop-in replacements for each other. That is deliberate - if you look at testing books over the years, there *are* different variations on equivalence classes, or techniques for testing on domain boundaries. So, the techniques presented here are suitable for testing *different* types of black-box requirements.

In practice, you are likely to have to use a variety of these techniques on any one program, based on the nature of each specific requirement. Some are going to have well-defined boundaries, so you can test on those. Often, you are going to have to have requirements that lead to categories and choices, so you can use the category-choice method there. And, sometimes, you are going to have too many test frames to be practicable, so you can choose to use pairwise (or n-wise) combinatoric testing to reduce the number of tests to a practical level.

So a test plan that says that you will use "domain testing" only, or "category-partition testing" only, is likely to be a bad one, because there will be requirements (either at the system level, or the detailed design level) that are best tested with other techniques.

My recommendation is that you recommend several techniques (these, and others that you may find in other references) and give guidelines as to when they should be applied. The rigor with which you apply formal techniques will probably vary depending on the sophistication of the project.

Given the informal nature of most requirements, it's unlikely that you will be able to automatically verify that a test suite has used your test strategies appropriately, so only human review by other experienced testers will be able to check that. However, you can combine black-box methods with white-box coverage criteria as an automatic adequacy check. We'll discuss this more later!

Summary

- Boundary value testing uses boundaries to determine the inputs and use them to identify tests
- Combinatorial testing makes combinations and pair wise testing is one of the famous types of combinatorial testing.
- We design test cases to execute all possible discrete combinations of each pair of input parameters in pairwise testing.
- Jenny tool is a shortcut to find test cases we can make using pairwise testing for complex systems.

References

1. https://www.guru99.com/black-box-testing.html
2. Software Testing: From Theory to Practice accessible at https://sttp.site/
3. Chapter 4 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.

4. Chapter 10 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.
5. Ostrand, T. J., & Balcer, M. J. (1988). The category-partition method for specifying and generating functional tests. Communications of the ACM, 31(6), 676-686.
6. Jeng, B., & Weyuker, E. J. (1994). A simplified domain-testing strategy. ACM Transactions on Software Engineering and Methodology (TOSEM), 3(3), 254-270.
7. http://burtleburtle.net/bob/math/jenny.html
8. http://www.pairwise.org/tools.asp
9.