Will the AND-1 riboswitch cleave itself when both of its OBS are bound?
Yes. By design, when both OBS are on TRUE state, the active hammerhead structure forms, which activates self-cleave.

Will the OR-1 riboswitch cleave itself when neither of its OBS are bound?
No. Either or both effector DNAs must be present for activating self-cleavage. If none of them are bound, the correct hammerhead structure won't form, failing to activate self-cleavage.

What behavior do we expect from the YES-1 riboswitch?
In the absence of 22-nd effector DNA, YES-1 is predicted to form the OFF-state structure (no self-cleavage), opposite in the presence of that. In its inactive formation, the nucleotides within the OBS are proposed to form a stem IV structure, which involves extensive base-pairing interaction with part of the hammerhead core and with most nucleotides that would otherwise form the stem II structure required for riboenzyme activation. When DNA effector and OBS are perfectly matched, self-cleavage activates, resulting a separation of cleavage fragment.

This trunk imports all necessary tools for later use.

In [1]:
```
import subprocess
import string
import sqlite3
import pandas as pd
from IPython.display import Image
conn = sqlite3.connect('my.db')
c = conn.cursor()
```

SQL TABLE

1) Manually enter the name, nucleotides, the first and second OBSs, and the first and second red regions. Assign these to corresponding variables. Gather them in a bigger list.
2) Create SQL table, use loop to access the lists within the bigger list, and change them to tuples so that they can be recongized by SQL.
3) Use pandas (package) to visualize the table.

In [80]:
```
YES_1 = ['YES_1', 'GGGCGACCCUGAUGAGCUUGAGUUUAGCUCGUCACUGUCCAGGUUCAAUCAGGC
NOT_1 = ['NOT_1', 'GGCAGGUACAUACAGCUGAUGAGUCCCAAAUAGGACGAAACGCGACACACACCA
AND_1 = ['AND_1', 'GGGCGACCCUGAUGAGCUUGGUUUAGUAUUUACAGCUCCAUACAUGAGGUGUUA
OR_1 = ['OR_1', 'GGGCGACCCUGAUGAGCUUGGUUGAGUAUUUACAGCUCCAUACUAGAGGUGUUCUC
all_data = [YES_1, NOT_1, AND_1, OR_1]
```

In [88]:
```
c.execute("""CREATE TABLE riboswitch (name, sequence, OBS1, OBS2, RED1, R
for _ in all_data:
    c.execute("""INSERT INTO riboswitch VALUES (?, ?, ?, ?, ?, ?);""", tu
c.execute("""SELECT * FROM riboswitch;""")
conn.commit()
```

In [89]:
```
print(c.fetchall())
```

```
[('YES_1', 'GGGCGACCCUGAUGAGCUUGAGUUUAGCUCGUCACUGUCCAGGUUCAAUCAGGCGAAA
CGGUGAAAGCCGUAGGUUGCCC', '26-47', 'NA', '16-21', '49-54'), ('NOT_1', '
GGCAGGUACAUACAGCUGAUGAGUCCCAAAUAGGACGAAACGCGACACACACCACUAAACCGUGCAGUGU
UUUGCGUCCUGUAUUCCACUGC', '44-66', 'NA', '40-43', '74-77'), ('AND_1', '
GGGCGACCCUGAUGAGCUUGGUUUAGUAUUUACAGCUCCAUACAUGAGGUGUUAUCCCUAUGCAAGUUCG
AUCAGGCGAAACGGUGAAAGCCGUAGGUUGCCCAGAGACAAU', '30-45', '49-64', '16-23'
, '70-77'), ('OR_1', 'GGGCGACCCUGAUGAGCUUGGUUGAGUAUUUACAGCUCCAUACUAGAG
GUGUUCUCCCUACGCAAGUUCGAUCAGGCGAAACGGUGAAAGCCGUAGGUUGCCC', '27-46', '47
-66', '16-26', '67-77')]
```

In [91]:
```
riboswitch_table = pd.read_sql("select * from riboswitch;", conn)
riboswitch_table
```

Out[91]:

	name	sequence	OBS1	OBS2	REI
0	YES_1	GGGCGACCCUGAUGAGCUUGAGUUUAGCUCGUCACUGUCCAGGUUC...	26-47	NA	16-
1	NOT_1	GGCAGGUACAUACAGCUGAUGAGUCCCAAAUAGGACGAAACGCGAC...	44-66	NA	40-
2	AND_1	GGGCGACCCUGAUGAGCUUGGUUUAGUAUUUACAGCUCCAUACAUG...	30-45	49-64	16-
3	OR_1	GGGCGACCCUGAUGAGCUUGGUUGAGUAUUUACAGCUCCAUACUAG...	27-46	47-66	16-

Create some functions that make later data manipulation easier.
The "get_xxx" functions access the data values.

In [7]:
```
def get_seq(data):
    return data[1]

def get_name(data):
    return data[0]

def get_OBS1(data):
    return data[2]

def get_OBS2(data):
    return data[3]

def get_RED1(data):
    return data[4]

def get_RED2(data):
    return data[5]
```

In [8]:
```
def unconstrain_positions(start, end): #A function that returns a list of
    num = []
    for n in range(start, end + 1):
        num.extend([n])
    return num
```

Note: These foldings do not contain any constrains (in other words, they are FALSE/FALSE).

> **subprocess.run** takes in terminal commends in list format, with each part separated. The **ascii()** method returns a string containing a printable representation of an object, which is our **input**. The **byte()** method returns a immutable bytes object initialized with he given size and data. The standard output ('**stdout**') is a file-like object. "**.decode()**" makes the strings used by the program understandable by users. "**Pipe**" allows us to visulize the data. "**stderr**" captures the error if any.
>
> There are two **output** files: 1) **rna.ps** and 2) **dot.ps** which are postscript files.

I used a loop to run through all riboenzymes and create ps files for each. I also used subprocess.call() to rename rna.ps and dot.ps to avoid new files being overwritten.

In [9]:
```
#ALL F/F
for rs in all_data:
    filename_ps = get_name(rs) + '_FF_rna' + '.ps'
    filename_dot = get_name(rs) + '_FF_dot' + '.ps'
    p = subprocess.run(['RNAfold', '-p'], input=bytes(get_seq(rs), 'ascii
    print(p.stdout.decode())
    subprocess.call(['mv', 'rna.ps', filename_ps])
```

```
subprocess.call([ 'mv' ,  'dot.ps' , filename_dot])
```

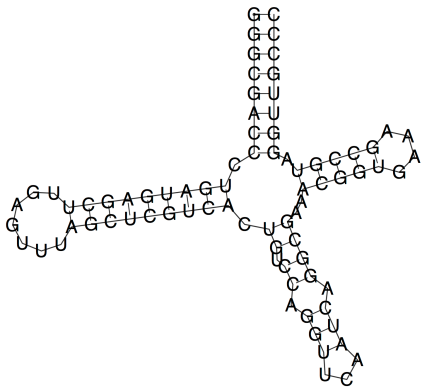
```
GGGCGACCCUGAUGAGCUUGAGUUUAGCUCGUCACUGUCCAGGUUCAAUCAGGCGAAACGGUGAAAGCCG
UAGGUUGCCC
(((((((.(((((((((.......)))))))))).((.((.(((...))).)))))..(((((....)))))
).)))))))) (-33.00)
(((((((.(((((((((.......)))))))))).{(.{{.{{{...,}).)}},..(((((....)))))
).)))))))) [-34.58]
(((((((.(((((((((.......)))))))))..(..............)...(((((....)))))
).)))))))) {-27.70 d=7.98}
 frequency of mfe structure in ensemble 0.0773552; ensemble diversity
12.30

GGCAGGUACAUACAGCUGAUGAGUCCCAAAUAGGACGAAACGCGACACACACCACUAAACCGUGCAGUGU
UUUGCGUCCUGUAUUCCACUGC
.((((....((((((.......(((......)))))...((((((.((((..(((......)))).))))
.))))))..))))))....)))) (-28.10)
.((((....((((((.......(((......)))))...((((((.((((..(((......)))).))))
.))))))..))))))....)))) [-28.67]
.((((....((((((.......(((......)))))...((((((.((((..(((......)))).))))
.))))))..))))))....)))) {-28.10 d=1.46}
 frequency of mfe structure in ensemble 0.396939; ensemble diversity 2
.74

GGGCGACCCUGAUGAGCUUGGUUUAGUAUUUACAGCUCCAUACAUGAGGUGUUAUCCCUAUGCAAGUUCG
AUCAGGCGAAACGGUGAAAGCCGUAGGUUGCCCAGAGACAAU
((((((((((((((((((...(((.....(((.(((.......))).))).....)))..)))))))).
)))))).....((((....))))).))))))).......... (-42.10)
((((((((((((((((((.,.(((.....(((.(((.......))).))).....))).,)))))),
}))))).....((((....))))).))))))).......... [-43.28]
((((((((((((((((((...(((.....(((.(((.......))).))).....)))..)))))))).
)))))).....((((....))))).))))))).......... {-42.10 d=2.98}
 frequency of mfe structure in ensemble 0.14638; ensemble diversity 4.
63

GGGCGACCCUGAUGAGCUUGGUUGAGUAUUUACAGCUCCAUACUAGAGGUGUUCUCCCUACGCAAGUUCG
AUCAGGCGAAACGGUGAAAGCCGUAGGUUGCCC
((((((((((((((((((.(((.....(((.(((.......))).))).)))...)).)))))))).
)))))).....((((....))))).)))))))) (-40.20)
((((((((((((((((((.{.{((.....(((.(((.......))).))).,,,.,,)).)))))),
}))))).....((((....))))).))))))).......... [-41.73]
((((((((((((((((((.........(((.(((.......))).))).......)).)))))))).
)))))).....((((....))))).))))))).......... {-39.40 d=4.70}
 frequency of mfe structure in ensemble 0.0839264; ensemble diversity
6.86
```

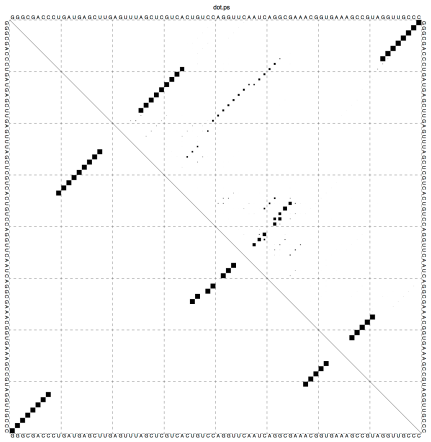
Each pair of brackets indicates a pair of matched base pairs i,j. The dots indicates the unmatches. {} and [] indicate weak pairs. The numbers are free energies (-kT ln(Z)) in kcal/mol.

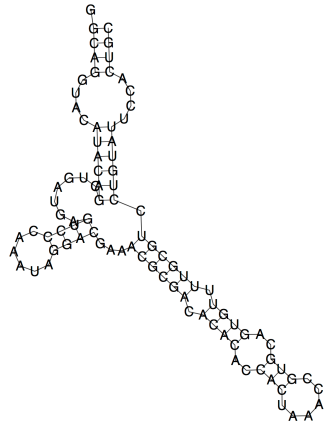
Generating one plot per riboswitch (YES-1, NOT-1, AND-1, and OR-1) - All FALSE

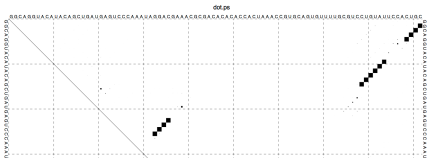
In [94]:
```
YFFR =Image(filename='YES_1_FF_rna.png', width=200, height=200)
YFFD =Image(filename='YES_1_FF_dot.png', width=200, height=200)
NFFR =Image(filename='NOT_1_FF_rna.png', width=200, height=200)
NFFD =Image(filename='NOT_1_FF_dot.png', width=200, height=200)
AFFR =Image(filename='AND_1_FF_rna.png', width=200, height=200)
AFFD =Image(filename='AND_1_FF_dot.png', width=200, height=200)
OFFR =Image(filename='OR_1_FF_rna.png', width=200, height=200)
OFFD =Image(filename='OR_1_FF_dot.png', width=200, height=200)
```

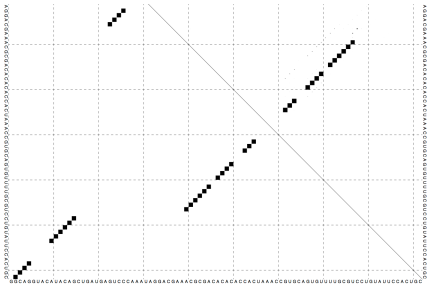
In [95]:
```
display(YFFR, YFFD, NFFR, NFFD, AFFR, AFFD, OFFR, OFFD)
```

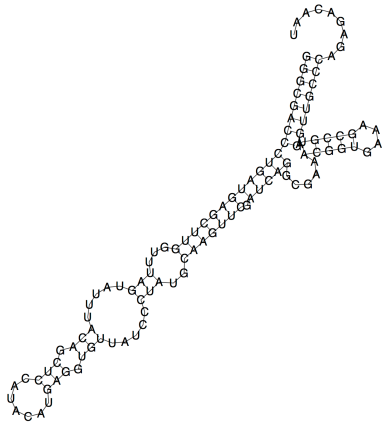


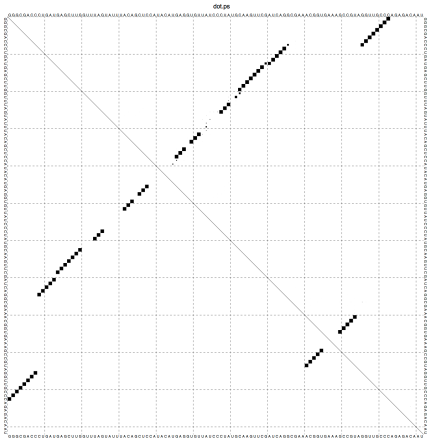


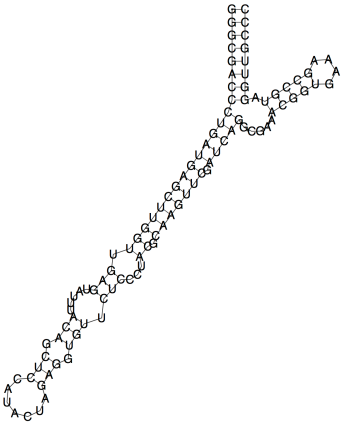


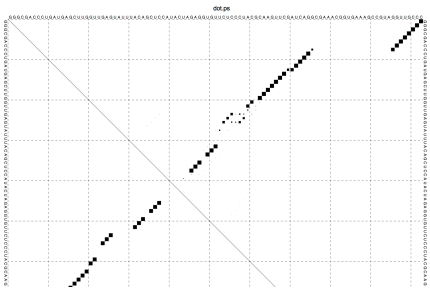


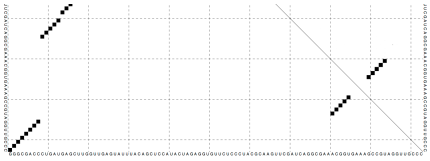





dot.ps




dot.ps



Compare each plot to the native conformation given in the publication. Are they the same? Are they different? Are there any stem-loop structures that don't match up? What might explain the differences? See if you can track down the parameters the authors used and compare them to the default RNAfold parameters (e.g., temperature, algorithm, etc)

The stem-loops structures produced here are not the same as those in the publication. The base pairs in the YES-1 here are different than those in the paper (red region involved in the loop). Also, the density of the dots in the dotplot is lighter than that in the paper (lower probability of forming that structure). NOT-1, OR-1 and AND-1 are the same here and publication. On page 1, the authors explain that they have used an algorithm "computes the entire ensemble of possible secondary structures as a function of temperature". Since E=(-kT ln(Z)) and the default temperture the RNAfold used here is 37C, temperature does affect the result of forming loops, which might cause the differences. Also, RNAfold keeps upgrading its algorithm, the versions of RNAfold might give different predictions.

In [10]:
```
def write_constrain(filename, riboswitch, cons_lst): #a function that writ
    f = open(filename, 'w')
    count = 1
    for nt in get_seq(riboswitch):
        f.write(nt)
    f.write('\n')
    for nt in get_seq(riboswitch):
        if count in cons_lst:
            f.write('x')
        else:
            f.write('.')
        count += 1
    f.close()
```

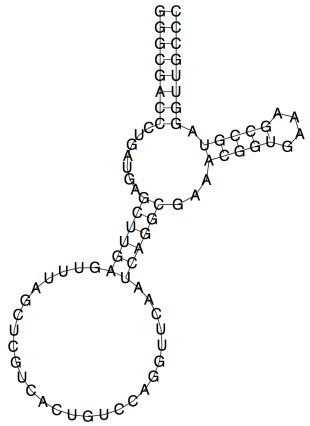
In [46]:
```
def run_RNAfold_C(txt_file, rename_ps, rename_dot): #run RNAfold and ren
    subprocess.run(['RNAfold', '-C', '-p', txt_file])
    subprocess.call(['mv', 'rna.ps', rename_ps])
    subprocess.call(['mv', 'dot.ps', rename_dot])
```

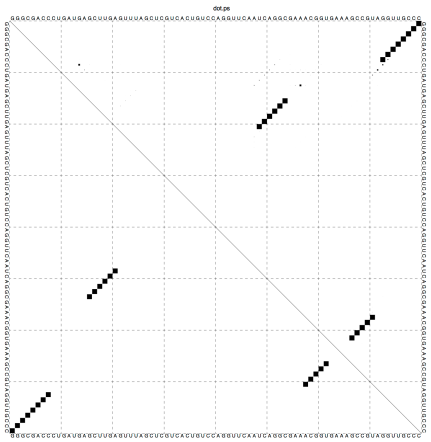
YES_1 and NOT_1

Note: These foldings do contain constrains (TRUE).

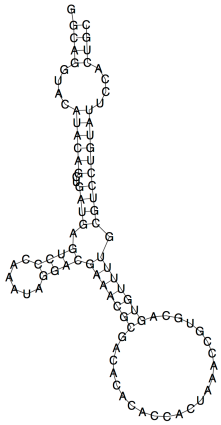
In [47]:
```
#write YES_1 and NOT_1 constrain files
YES_1_T = unconstrain_positions(26, 47)
NOT_1_T = unconstrain_positions(44, 66)
write_constrain('YES_1_T.txt', YES_1, YES_1_T)
write_constrain('NOT_1_T.txt', NOT_1, NOT_1_T)

run_RNAfold_C('YES_1_T.txt', 'YES_1_T_rna.ps', 'YES_1_T_dot.ps')
run_RNAfold_C('NOT_1_T.txt', 'NOT_1_T_rna.ps', 'NOT_1_T_dot.ps')
```

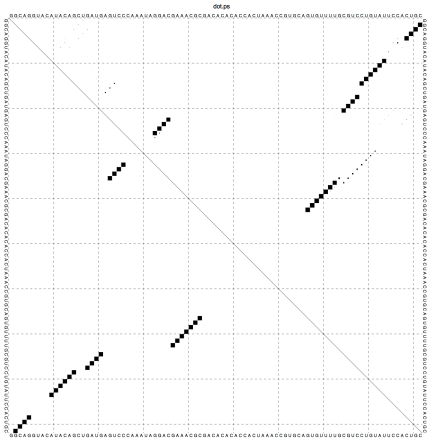
In [96]:
```
YTR =Image(filename='YES_1_T_rna.png', width=200, height=200)
YTD =Image(filename='YES_1_T_dot.png', width=200, height=200)
display(YTR, YTD)
```





In [98]:
```
NTR =Image(filename='NOT_1_T_rna.png', width=200, height=200)
NTD =Image(filename='NOT_1_T_dot.png', width=200, height=200)
display(NTR, NTD)
```





Does it look like the self-cleaving form of YES-1 in Figure 2? Are the red regions bound to each other?

Both YES_1 and NOT_1 look like the self-cleaving forms in the publication, so do the red regions. This makes sense because the constrain is set here. These constrains allow the loops to "form" hammerhand structures we desired, which enable self-cleavage.

AND_1

Note: These foldings implement these binary logic gates: TRUE/FALSE, FALSE/TRUE, TRUE/TRUE
The unconstrain positions are assigned to their corresponding variables.
Use the functions I wrote above to write txt files and run RNAfold for each riboenzyme.

In [48]:
```
#write AND_1 constrain files and RNAfold files
AND_1_TF = unconstrain_positions(30, 45)
AND_1_FT = unconstrain_positions(49, 64)
AND_1_TT = AND_1_TF + AND_1_FT

write_constrain('AND_1_TF.txt', AND_1, AND_1_TF)
write_constrain('AND_1_FT.txt', AND_1, AND_1_FT)
write_constrain('AND_1_TT.txt', AND_1, AND_1_TT)

run_RNAfold_C('AND_1_TF.txt', 'AND_1_TF_rna.ps', 'AND_1_TF_dot.ps')
run_RNAfold_C('AND_1_FT.txt', 'AND_1_FT_rna.ps', 'AND_1_FT_dot.ps')
run_RNAfold_C('AND_1_TT.txt', 'AND_1_TT_rna.ps', 'AND_1_TT_dot.ps')
```

OR_1

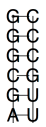
Note: These foldings implement these binary logic gates: TRUE/FALSE, FALSE/TRUE, TRUE/TRUE

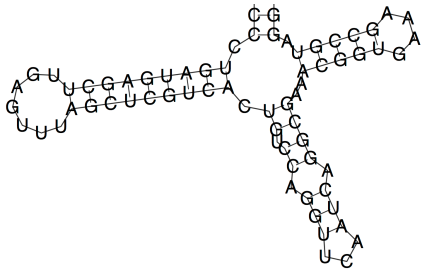
In [49]:
```
#write OR_1 constrain files and RNAfold files

OR_1_TF = unconstrain_positions(27, 46)
OR_1_FT = unconstrain_positions(47, 66)
OR_1_TT = OR_1_TF + OR_1_FT

write_constrain('OR_1_TF.txt', OR_1, OR_1_TF)
write_constrain('OR_1_FT.txt', OR_1, OR_1_FT)
write_constrain('OR_1_TT.txt', OR_1, OR_1_TT)

run_RNAfold_C('OR_1_TF.txt', 'OR_1_TF_rna.ps', 'OR_1_TF_dot.ps')
run_RNAfold_C('OR_1_FT.txt', 'OR_1_FT_rna.ps', 'OR_1_FT_dot.ps')
run_RNAfold_C('OR_1_TT.txt', 'OR_1_TT_rna.ps', 'OR_1_TT_dot.ps')
```

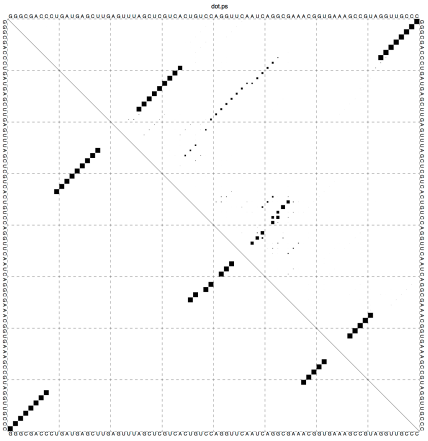
ALL RESULTS

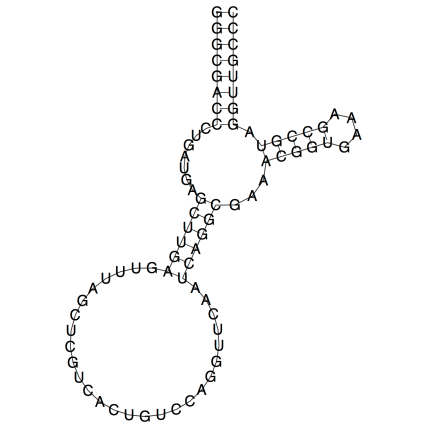
YES_1

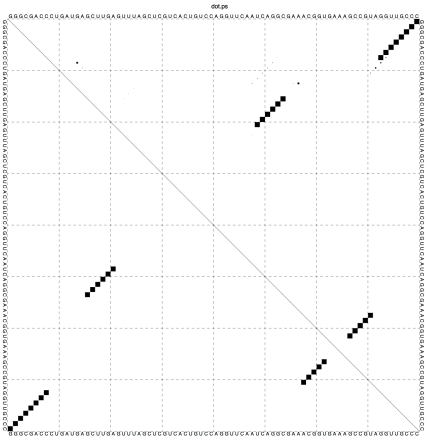
In [53]:
```
YFFR =Image(filename='YES_1_FF_rna.png', width=200, height=200)
YFFD =Image(filename='YES_1_FF_dot.png', width=200, height=200)

YTR =Image(filename='YES_1_T_rna.png', width=200, height=200)
YTD =Image(filename='YES_1_T_dot.png', width=200, height=200)

display(YFFR, YFFD, YTR, YTD)
```







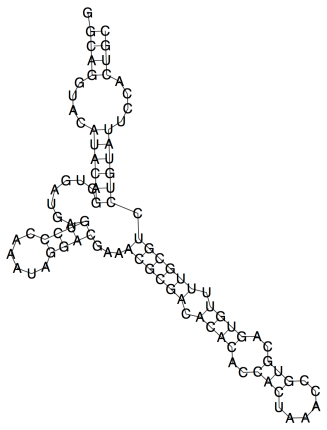


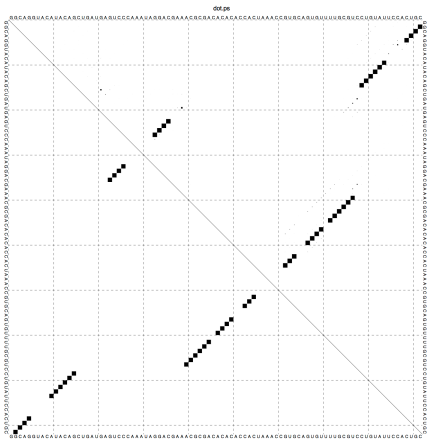


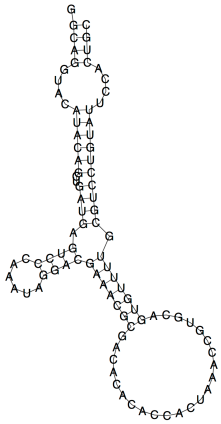
NOT_1

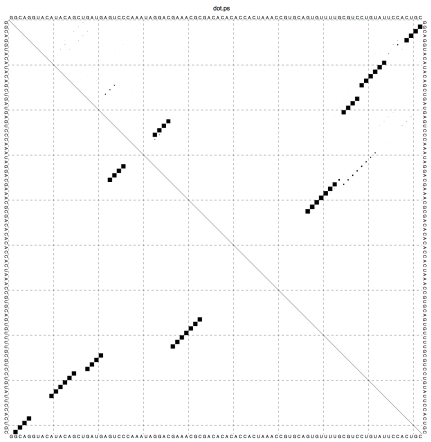
In [97]:
```
NFFR =Image(filename='NOT_1_FF_rna.png', width=200, height=200)
NFFD =Image(filename='NOT_1_FF_dot.png', width=200, height=200)
NTR =Image(filename='NOT_1_T_rna.png', width=200, height=200)
```

```
NTD =Image(filename= 'NOT_1_T_dot.png' , width=200, height=200)

display(NFFR, NFFD, NTR, NTD)
```



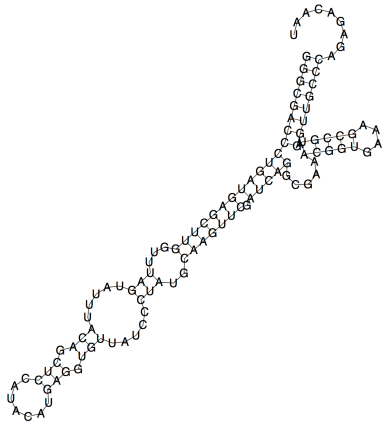


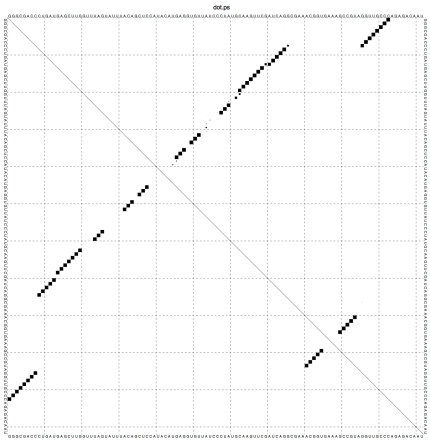


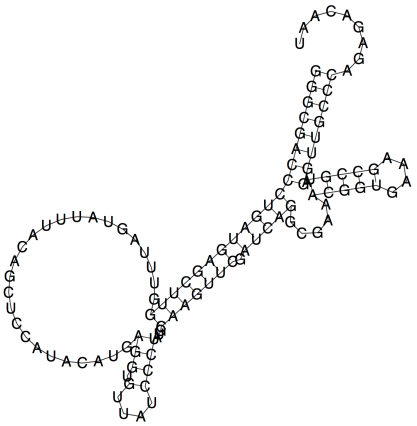


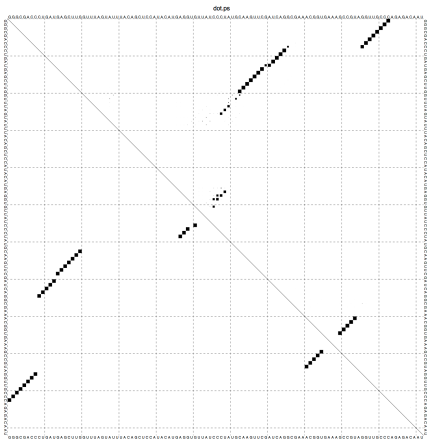
AND_1

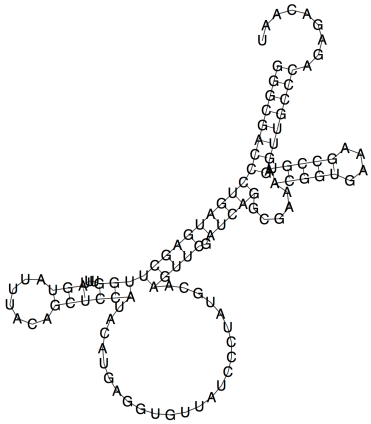
In [99]:
```
AFFR =Image(filename='AND_1_FF_rna.png', width=200, height=200)
AFFD =Image(filename='AND_1_FF_dot.png', width=200, height=200)

ATFR1 =Image(filename='AND_1_TF_rna.png', width=200, height=200)
ATFD1 =Image(filename='AND_1_TF_dot.png', width=200, height=200)

ATFR2 =Image(filename='AND_1_FT_rna.png', width=200, height=200)
ATFD2 =Image(filename='AND_1_FT_dot.png', width=200, height=200)

ATTR =Image(filename='AND_1_TT_rna.png', width=200, height=200)
ATTD =Image(filename='AND_1_TT_dot.png', width=200, height=200)
display(AFFR, AFFD, ATFR1, ATFD1, ATFR2, ATFD2, ATTR, ATTD)
```

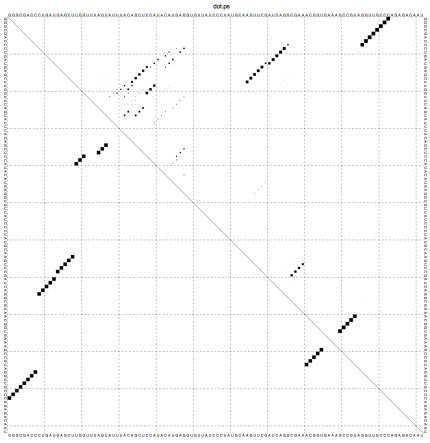


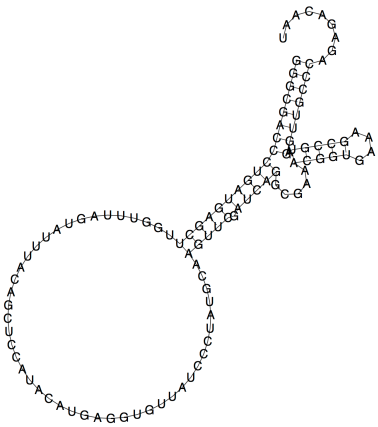


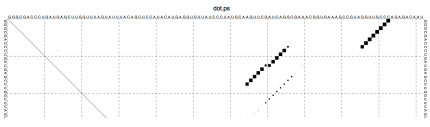


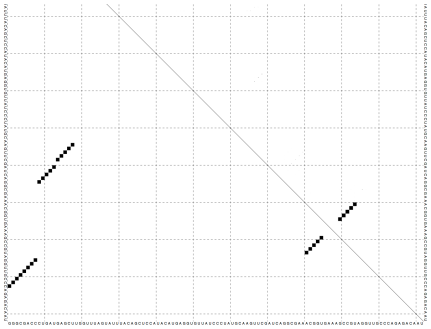

dot.ps




dot.ps

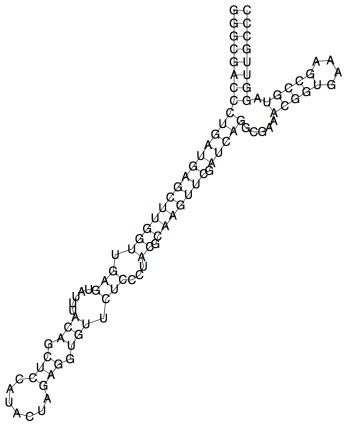


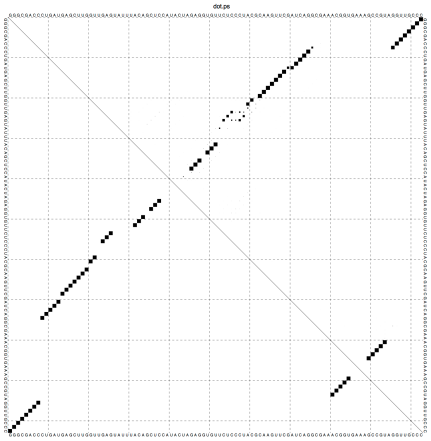

dot.ps

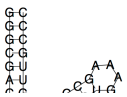


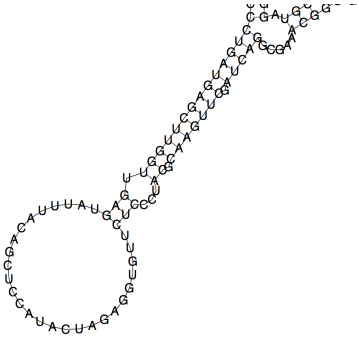
OR_1

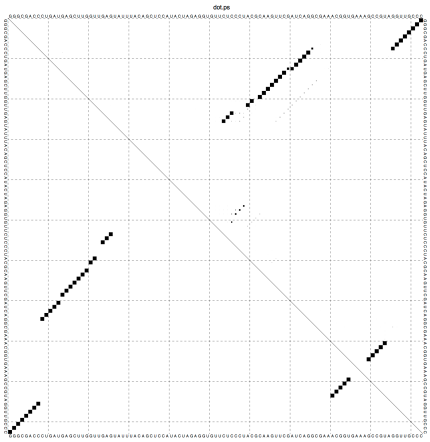
In [100]:
```
OFFR =Image(filename='OR_1_FF_rna.png', width=200, height=200)
OFFD =Image(filename='OR_1_FF_dot.png', width=200, height=200)

OTFR1 =Image(filename='OR_1_TF_rna.png', width=200, height=200)
OTFD1 =Image(filename='OR_1_TF_dot.png', width=200, height=200)

OTFR2 =Image(filename='OR_1_FT_rna.png', width=200, height=200)
OTFD2 =Image(filename='OR_1_FT_dot.png', width=200, height=200)

OTTR =Image(filename='OR_1_TT_rna.png', width=200, height=200)
OTTD =Image(filename='OR_1_TT_dot.png', width=200, height=200)
display(OFFR, OFFD, OTFR1, OTFD1, OTFR2, OTFD2, OTTR, OTTD)
```

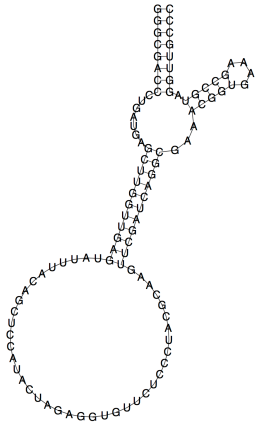


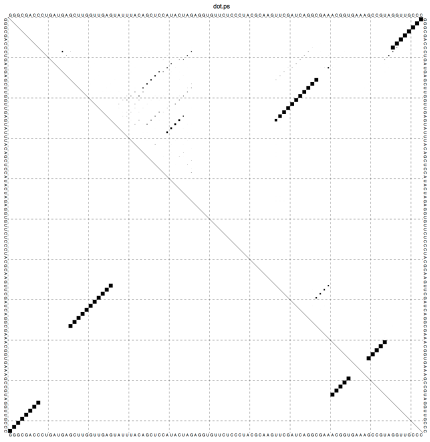


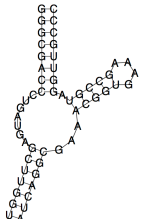


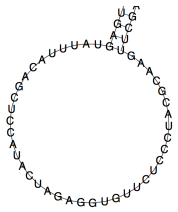


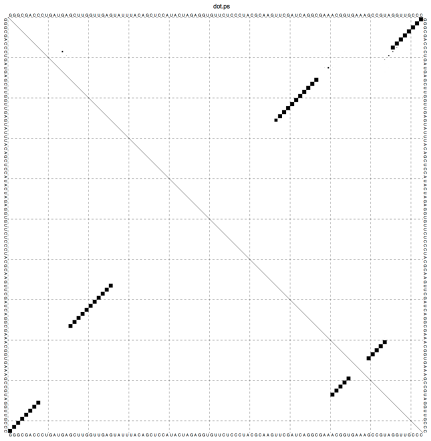

dot.ps




dot.ps







Truth Table

In [103]:
```
c.execute("""CREATE TABLE truth (riboswitch, NONE, OBS1, OBS2, OBS1and2
c.execute("""INSERT INTO truth VALUES ("AND_1", "false", "false", "false
                                      ("OR_1", "false", "true", "true",
conn.commit()
```

In [104]:
```
truth_table = pd.read_sql("select * from truth;", conn)
truth_table
```

Out[104]:

	riboswitch	NONE	OBS1	OBS2	OBS1and2
0	AND_1	false	false	false	true
1	OR_1	false	true	true	true

According to your results, do the AND-1 and OR-1 riboswitches work as the paper claims?

Yes. By looking at the shape, in AND_1, both constrains (OBS) must be present in order to enable self-cleavage, as shown in the last graph of AND_1. When either OBS is absent, the loops are twisted because part of the sequence is not constrained. In OR_1, the presence of either constrain can induce self-cleavage form, the correct hammerhand structure. As shown in the last 3 graphs of OR_1, the basepairs are exactly the same, forming the loops the paper claims.

In [ ]: