Begin Modern Programming with Python

Pyi Soe

အခန်း ၁

ကားရဲလ်နှင့် ရီကားဆစ်ဖ် ဖန်ရှင်များ

ဖန်ရှင်တစ်ခုကနေ 'အခြား' ဖန်ရှင်တွေ ခေါ်သုံးတာကို အခန်း (၃) မှာ တွေ့ခဲ့ပြီးပါပြီ။ ဒါပေမဲ့ ဖန်ရှင်တစ်ခုက ၎င်းကိုယ်၎င်း ပြန်ခေါ်ထားတာကိုတော့ မကြုံဖူးသေးပါဘူး။ ဖန်ရှင်တွေဟာ ဆော့ဖ်ဝဲ အဆောက်အဦး တည်ဆောက်ရာမှာ မရှိမဖြစ်တဲ့ အခြေခံအုတ်ချပ်တွေလို့ ဆိုရမှာပါ။ ၎င်းကိုယ်တိုင်ကို ပြန်လည်အသုံးပြု၍ ဖန်ရှင်အုတ်ချပ်တစ်ခု ဖန်တီးလို့ ရနိုင်ပါမလား။ ဒီမေးခွန်းဟာ ထူးဆန်းကောင်း ထူးဆန်းနေပါလိမ့်မယ်။ အခြေခံကျပြီး စိတ်ဝင်စားစရာကောင်းတဲ့ ဖီလော်ဆော်ဖီ မေးခွန်းလည်း ဖြစ်တယ်။

ဖန်ရှင် သတ်မှတ်ချက်ထဲမှာ ၎င်းဖန်ရှင်ကိုယ်တိုင်ကို ပြန်ခေါ်လို့ ရပါတယ်။ ရီကားဆစ်ဖ် ဖန်ရှင် (*recursive function*) လို့ ခေါ်တယ်။ ရီကားဆစ်ဖ် ဖန်ရှင်တွေဟာ ဘီဂင်နာ ပရိုဂရမ်မာအတွက် နားလည်ဖို့ ခက်ခဲတဲ့ သဘောတရားအဖြစ် ယူဆကြတာကြောင့် စာအုပ် အတော်များများမှာ နောက်ကျပြီး ဖော်ပြလေ့ရှိတယ်။ တကယ်က လူများစု ထင်/ပြောသလို နားမလည်နိုင်လောက်အောင် ရှုပ်ထွေး ခက်ခဲတဲ့ သဘောတရား မဟုတ်ပါဘူး။ သာမန်လူအားလုံး နားလည်နိုင်ပါတယ်။ ဒါကြောင့် စောစောစီးစီး အခုပဲ မိတ်ဆက်ပေးလိုက်ပါတယ်။ အကယ်၍ နားမလည်ခဲ့ရင်လည်း ပြဿနာမရှိပါဘူး။ အကြမ်းဖျဉ်းလောက် ဖတ်ကြည့်ပြီး နောက်လာမဲ့ အခန်းတွေကို ကျော်သွားနိုင်ပါတယ်။ နောင်တစ်ချိန်ကျမှ ပြန်လာဖတ်ပေါ့။

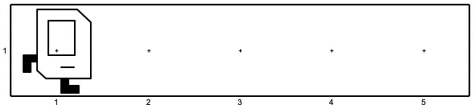
၁.၁ ရီကားဆစ်ဖ် ဖန်ရှင် ဘယ်လို အလုပ်လုပ်လဲ

ရီကားဆစ်ဖ် ဖန်ရှင် ဥပမာတစ်ခုကို လေ့လာကြည့်ပါမယ်။ အောက်ဖော်ပြပါ ဖန်ရှင်ဟာ ၎င်းကိုယ်တိုင်ကို ၎င်း ပြန်ခေါ်ထားတာ ဂရုပြုကြည့်ပါ။ *recursive call* လို့ ကွန်းမန့်ရေးထားတဲ့ လိုင်းမှာပါ။

```
def make_beeper_row():
    if front_is_clear():
        put_beeper()
        move()
        make_beeper_row() # recursive call
    else:
        put_beeper()
```

ဒီဖန်ရှင်ကို ခေါ်လိုက်ရင် ဘာဆက်ဖြစ်မလဲဆိုတာ စိတ်ဝင်စားစရာပါ။ ဖန်ရှင်မစတင်မီ အနေအထားကို ပုံ (၁.၁) မှာ ကြည့်ပါ။ ဖန်ရှင်ကို ကနဦး စခေါ်လိုက်တာမို့လို့ *initial call* လို့ ရည်ညွှန်းပါမယ်။

```
# initial call
make_beeper_row()
```



ပုံ ၁.၁

ဖန်ရှင် စတင် လုပ်ဆောင်ပါမယ်။ ရှေ့မှာရှင်းနေတဲ့ အတွက် `if` ဘလောက်ကို လုပ်မှာပါ

```
put_beeper()
move()
make_beeper_row() # recursive call
```

ဘိပါချ၊ ရှေ့တိုး [[ပုံ (၁.၂) (က) ကြည့်ပါ]] ပြီးရင် သူ့ကိုယ်သူ ပြန်ခေါ်ထားတယ်။ ဒါဟာ ပထမဆုံး တစ်ကြိမ်ပါ။ ဖန်ရှင်ခေါ်ရင် ဖြစ်မြဲအတိုင်းပဲ ဖန်ရှင်နဲ့ သက်ဆိုင်တဲ့ ဘလောက်ကို ဆောက်ရွက်တာပေါ့။ ဒီတော့ `make_beeper_row` ဖန်ရှင်ဘလောက်ကိုပဲ တစ်ခါထပ်လုပ်မှာပါ။ ရှေ့မှာ ရှင်းနေတဲ့အတွက် `if` အပိုင်းကို လုပ်တယ်

```
put_beeper()
move()
make_beeper_row() # recursive call
```

ဘိပါချ၊ ရှေ့တိုး [[ပုံ (၁.၂) (ခ)]] ပြီး သူ့ကိုယ်သူ ပြန်ခေါ်ထားတဲ့ကိစ္စ တစ်ခါထပ်ဖြစ်ပြန်တယ်။ ဒါနဲ့ဆို နှစ်ကြိမ်။ ဖန်ရှင်ဘလောက် အလုပ် ပြန်လုပ်မယ်။ ရှေ့မှာရှင်းတယ်၊ `if` ကိုပဲ ထပ်လုပ်

```
put_beeper()
move()
make_beeper_row() # recursive call
```

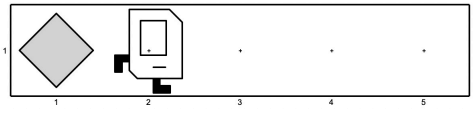
ပုံ (၁.၂) (ဂ) နေရာရောက်ပြီး သူ့ကိုယ်သူ ထပ်ခေါ်ထားပြန်တယ်။ သုံးကြိမ်ရှိပြီ။ ဒီတစ်ခါလည်း `if` အပိုင်းပဲ ထပ်လုပ်

```
put_beeper()
move()
make_beeper_row() # recursive call
```

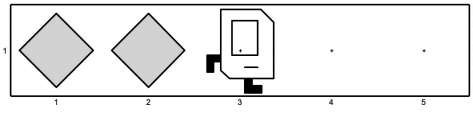
ရှေ့တိုးပြီးရင် နံရံပိတ်နေပြီ [[ပုံ (၁.၂) (ဃ)]]။ သူ့ကိုယ်သူ ခေါ်တယ်။ ရှေ့မှာ ပိတ်နေတဲ့အတွက် `else` အပိုင်းကို လုပ်မှာပါ။

```
put_beeper()
```

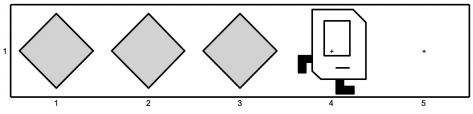
သူ့ကိုယ်သူ ပြန်ခေါ်တဲ့ကစ္စ ထပ်မဖြစ်တော့ဘူး။ ဒီမှာပဲ ပြီးဆုံးသွားတယ်။ ရီကားဆစ်ဖ် ဖန်ရှင် အလုပ် လုပ်ပုံ အခြေခံသဘောတရားက ဒါပါပဲ။ loop တွေ မသုံးဘဲ ပြန်ကျော့နေတဲ့ သဘောကို ရီကားဆစ်ဖ် ဖန်ရှင်မှာ တွေ့ရပါတယ်။ ဖန်ရှင်က သူ့ကိုယ်သူ (သို့ ကိုယ့်ကိုကိုယ်) ပြန်ခေါ်တာကို *recursive call* လို့



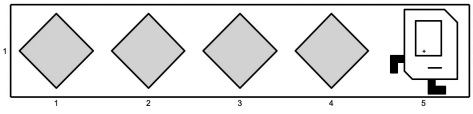
(က) ပထမ တစ်ကျော့ပြီး



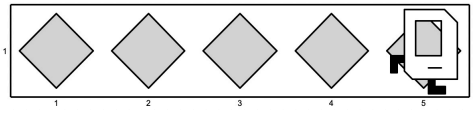
(ခ) ဒုတိယ တစ်ကျော့ပြီး



(ဂ) တတိယ တစ်ကျော့ပြီး



(ဃ) စတုတ္ထမြောက် ကျော့ပြီး



(င) နောက်ဆုံး `putBeeper` လုပ်ပြီး

ပုံ ၁.၂

ခေါ်ပါတယ်။ ဒက်ဖ်နေးရှင်းအရ recursive function တွေမှာ recursive call အနည်းဆုံး တစ်ခု ပါရမှာ ဖြစ်ပါတယ်။

```
def make_beeper_row():
    if front_is_clear():
        put_beeper()
        move()
        make_beeper_row()
    else:
        put_beeper()

make_beeper_row()
```

ရီကားဆစ်ဖ်ကောလ် ဖြစ်တာကို စိတ်ကူးပုံဖော်ကြည့်ဖို့ ပြထားတာပါ။ အပြင်ဆုံး မြှားအနက်က ကနဦး ဖန်ရှင်ကောလ် စတင်တာဖြစ်ပေါ်တာကို ဖော်ပြတယ်။ ရီကားဆစ်ဖ်ကောလ် မဟုတ်သေးဘူး။ မြှားအပြာက ရီကားဆစ်ဖ်ကောလ်ကြောင့် ဖန်ရှင်အစ ပြန်ရောက်သွားတာကို ပြတယ်။ ပထမနဲ့ ဒုတိယ ရီကားဆစ်ဖ်ကောလ် နှစ်ခုအတွက်ပြထားတာပါ။ ရှေ့က ဥပမာအတွက် မြှားအပြာ လေးခု ရှိရမှာပါ (ရီကားဆစ်ဖ်ကောလ် လေးကြိမ်အတွက်)။ မြှားအပြာ နောက်ထပ် နှစ်ခုရှိတယ် မှတ်ယူပါ (ပုံမှာထပ်ထည့်ရင် ကြပ်ညပ်ပြီး ကြည့်ရရှုပ်လို့ မဆွဲပြတာ)။ လေးကြိမ်မြောက်မှာ ရီကားဆစ်ဖ်ကောလ် ထပ်မဖြစ်တော့ဘူး (`if` အပိုင်းကို မလုပ်တော့တဲ့အတွက်)။ ဘိပါချုပြီး ဖန်ရှင်ကောလ် စခဲ့တဲ့နေရာကို ပြန်ရောက် သွားမယ် (မြှားအနီ)။ ဘယ်လို ပြန်ရောက်သွားတာလဲ ဆက်ကြည့်ရအောင်။

နောက်ဆုံး ရီကားဆစ်ဖ်ကောလ်မှ ပြန်လာခြင်း

နောက်ဆုံး ရီကားဆစ်ဖ်ကောလ်ကနေ မူလ ဖန်ရှင်ခေါ်ခဲ့တဲ့ နေရာကို ဘယ်လိုပြန်ရောက်သွားတာလဲ။ ဒီကိစ္စနားလည်ဖို့ ဖန်ရှင် *return* ပြန်ခြင်းအကြောင်း အရင်ကြည့်ရပါမယ်။ ဖန်ရှင်ကောလ် လုပ်ဆောင်တဲ့အခါ အဲဒီဖန်ရှင်နဲ့ သက်ဆိုင်တဲ့ ဘလောက်ဆီကို ခုန်ကျော် ရောက်ရှိသွားမှာပါ။ ဖန်ရှင်ဘလောက်ကို လုပ်ဆောင်ပြီး ခေါ်ခဲ့တဲ့ နေရာကို ပြန်လည်ရောက်ရှိသွားမှာ ဖြစ်တယ်။ ဒီဖြစ်စဉ်ကို ဖန်ရှင် return ပြန်တယ်လို့ ပြောပါတယ်။

```
def main():
    turn_right()
    move()
    turn_right()
    move()

def turn_right():
    turn_left()
    turn_left()
    turn_left()
```

ပထမ turn_right ကောလ် လုပ်ဆောင်တဲ့အခါ ကောလ်လုပ်တဲ့ နေရာကနေ turn_right ဖန်ရှင်ထဲကို jump လုပ်ပြီး ရောက်သွားတယ်။ မြှားအနက်နဲ့ ပြထားတယ်။ ဖန်ရှင်ဘလောက် လုပ်ဆောင်ပြီးတဲ့အခါ ခေါ်ခဲ့တဲ့နေရာ main ဖန်ရှင်ထဲ ပြန်ရောက်သွားတယ် (မြှားအနီ)။ ဒုတိယ turn_right လည်း ထိုနည်းတူစွာပဲ ဖြစ်ပါတယ်။

```
def main():
    turn_right()
    move()
    turn_right()
    move()

def turn_right():
    turn_left()
    turn_left()
    turn_left()
```

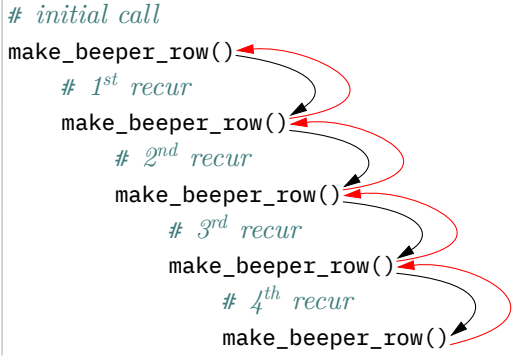
နှစ်ဆင့်၊ သုံးဆင့် ဖန်ရှင်ကောလ်တွေမှာလည်း ဒီသဘောတရား အတိုင်းပါပဲ။ အောက်ဖော်ပြပါ ပရိဂရမ်ကုဒ်ကို ကြည့်ပါ။ main ဖန်ရှင်ထဲကနေ do_tricks ဆီကို ရောက်သွားမယ်။ do_tricks ထဲကနေ put_two ထဲကို ရောက်သွားမယ်။

```
def main():
    do_tricks()
    move()

def do_tricks():
    move()
    put_two()
    turn_left()
    move()

def put_two():
    put_beeper()
    put_beeper()
```

`put_two` ပြီးသွားတဲ့အခါ `do_tricks` ထဲ ပြန်ရောက်သွားမယ်။ `turn_left`, `move` ဆက်လုပ်ပြီး `do_tricks` ခေါ်ခဲ့တဲ့နေရာ `main` ထဲ ပြန်ရောက်သွားမယ်။ နောက်ဆုံး `main` ထဲက `move` ကို ဆက်လုပ်ပါတယ်။

ဖန်ရှင် အဆင့်ဆင့် ခေါ်ထားတဲ့အခါ နောက်ဆုံးခေါ်တဲ့ ဖန်ရှင်က အရင်ဆုံး return ပြန်ပါတယ်။ `main` ကနေ `do_tricks` ကိုခေါ်၊ `do_tricks` ကနေ `put_two` ကိုခေါ်ထားရင် `put_two` ကနေ `do_tricks` ဆီကို အရင် return ပြန်တယ်။ ပြီးတော့မှ `do_tricks` ကနေ `main` ကို ပြန်ရောက်မှာပါ။ ဒီသဘောအရ `put_two` return မပြန်မချင်း `do_tricks` ဖန်ရှင်မပြီးသေးဘူး။ `put_two` ကနေ ပြန်လာပြီးမှ ကျန်တဲ့ `turn_left`, `move` ဆက်လုပ်တယ်။ ပြီးတော့မှ `do_tricks` ဖန်ရှင်က return ပြန်ပါတယ်။

ရီကားဆစ်ဖ် ဖန်ရှင်ကောလ်တွေ ဘယ်လို return ပြန်လဲ။ ရှေ့ကလို မြှားတွေနဲ့ ဆွဲပြလို့ ရပေမဲ့ ကြည့်ရတာ ရှုပ်ရှက်ခတ်နေမှာပါ။ အခုလို မြင်ကြည့်ရင် ပိုရှင်းပါတယ်။



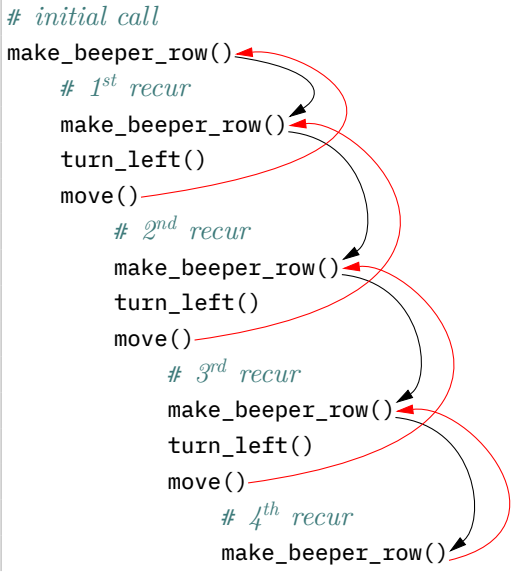
မြှားအနက်တွေက ဖန်ရှင်ကောလ် တစ်ဆင့်ပြီးတစ်ဆင့် ဖြစ်တာကို ပြတာပါ။ အထက်မှအောက် အစီအစဉ်အတိုင်း ဖြစ်ပါတယ်။ လေးကြိမ်မြောက်မှာ နောက်ထပ် ရီကားဆစ်ဖ်ကောလ် ထပ်မဖြစ်တော့ဘဲ နောက်ဆုံး ရီကားဆစ်ဖ်ကောလ်က အရင်ဆုံး return စပြန်ပါတယ် (အောက်ဆုံး မြှားအနီနဲ့ ပြထား)။ ဒီအခါ တတိယ ရီကားဆစ်ဖ်ကောလ်ကို ပြန်ရောက်သွားမှာပါ။ ဒီအတိုင်း တစ်ဆင့်ပြီးတစ်ဆင့် အထက်ကို return ပြန်ပြီး နောက်ဆုံးမှာ ပထမဆုံး ခေါ်ခဲ့တဲ့နေရာ ပြန်ရောက်သွားမှာပါ။ (အခုပြထားတာကို တကယ့် Python ကုဒ် အနေနဲ့ မယူဆရပါ၊ ဖြစ်စဉ် နားလည်အောင် ပြခြင်းသာဖြစ်ပါတယ်)။ နဲနဲပြင်ထားတဲ့ `make_beeper_row` ဗားရှင်းမှာ return ပြန်တဲ့ ဖြစ်စဉ်ကို ကြည့်ရအောင်။

```
make_beeper_row()
```

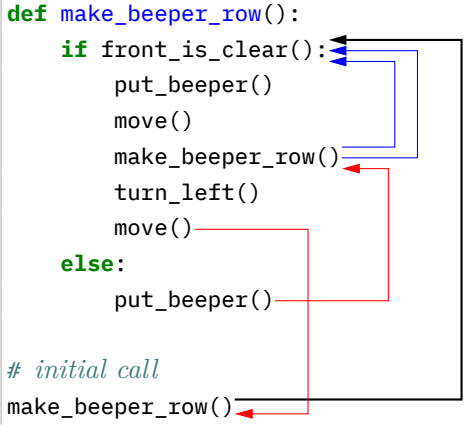
```
def make_beeper_row():
    if front_is_clear():
        put_beeper()
        move()
        make_beeper_row()
        turn_left()
        move()
    else:
        put_beeper()
```

`if` အပိုင်း ရီကားဆစ်ဖ်ကောလ်ပြီး `turn_left` နဲ့ `move` ထပ်ဖြည့်ထားတာ။ ရီကားဆစ်ဖ်ကောလ် return ပြန်လာပြီးမှ ဒီနှစ်ခု ဆက်လုပ်မှာပါ။ နောက်ဆုံး ရီကားဆစ်ဖ်ကောလ်က return စဖြစ်တယ်။ ဒီ

တော့မှ တတိယအောက် `turn_left` နဲ့ `move` ကို လုပ်ဆောင်မှာပါ။ ပြီးမှ တတိယကောလ် return ပြန်တယ်။ ဒီအတိုင်း အထက်ကို တက်သွားပြီး ကျန်နေသေးတဲ့ စတိတ်မန့်တွေကို လုပ်ဆောင်ပါတယ်။ ပထမဆုံးကောလ်အောက် ကျန်နေတာတွေ နောက်ဆုံးကျမှ ပြီးမှာပါ။



ရီကားဆစ်ဖ်ကောလ် နှစ်ခါပဲ ဖြစ်မယ်ဆိုရင် အောက်ပါအတိုင်း မြင်ကြည့်လို့ ရပါတယ်။ မြှားအပြာနှစ်ခုက ရီကားဆစ်ဖ်ကောလ်ဖြစ်တာ။ ဒုတိယကောလ်က ဘိပါချပြီး (`else` အပိုင်း) အရင် return မယ်။ အထက်ကို ညွှန်တဲ့ မြှားအနီကို ကြည့်ပါ။ ဘယ်လှည့်၊ ရှေ့တိုးပြီး ပထမ ကောလ်က နောက်မှ initial call လုပ်ခဲ့တဲ့ဆီ ပြန်ရောက်တာ။ အောက်ကိုညွှန်တဲ့ မြှားအနီကို ကြည့်ပါ။

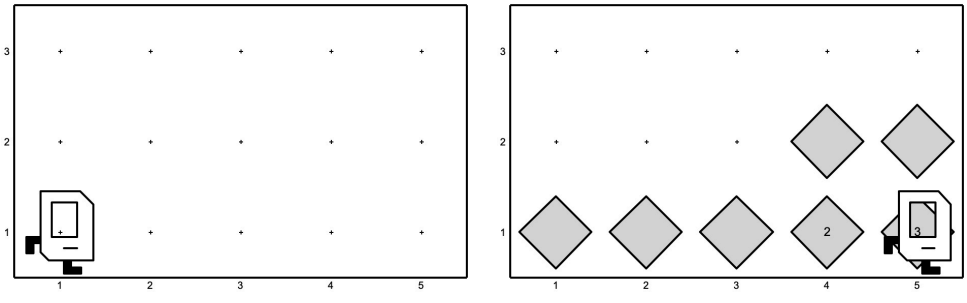


ရီကားဆစ်ဖ် ဖန်ရှင်အကြောင်း လေ့လာတဲ့အခါ ဘီဂင်နာအများစု ကြေကြေညက်ညက် နားလည်ဖို့ အတွက် အခက်အခဲဆုံးတစ်ခုက return ပြန်တဲ့ သဘောတရားပါပဲ။ များများစဉ်းစား၊ များများလေ့ကျင့်ရင် ဒီအခက်အခဲ ကျော်ဖြတ်နိုင်မှာပါ။ `if...else` အပြီးမှာ `put_beeper` လေးတစ်ခုပဲ ထပ်ဖြည့်လိုက်ရင် ဘယ်လိုဖြစ်မလဲ။

```
# File: make_beeper_row2.py
def make_beeper_row():
```

```
    if front_is_clear():
        put_beeper()
        move()
        make_beeper_row()
        turn_left()
        move()
    else:
        put_beeper()
    put_beeper()

# initial call
make_beeper_row()
```

ပုံ (၁.၃) (က) နဲ့ (ခ) က မတိုင်မီနဲ့ ပြီးနောက် အခြေအနေပါ။ နောက်ဆုံးမှာ ဘိပါလေးခုကို ပါတ်လည် ဘယ်လိုချသွားလဲ စဉ်းစားကြည့်ပါ။ ရှေ့မှာဖော်ပြခဲ့သလို ဖန်ရှင်ကောလ်တွေ တစ်ဆင့်ပြီးတစ်ဆင့် ဖြစ်ပုံနဲ့ return ဖြစ်ပုံကို မြှားဆွဲကြည့်ပါ။



(က) (ခ)

ပုံ ၁.၃

၁.၂ ရီကားဆစ်ဖ်နည်းဖြင့် ပုဏ္ဍာဖြေရှင်းခြင်း

ရှေ့ စက်ရှင်မှာ လေ့လာခဲ့တာက ရီကားဆစ်ဖ်ဖန်ရှင် ဘယ်လို အလုပ်လုပ်လဲပဲ ရှိပါသေးတယ်။ တစ်နည်း အားဖြင့် မက္ကနစ်ဇမ် (mechanism) ကို လေ့လာတာပါ။ အခုတစ်ခါ ရီကားဆစ်ဖ်ဖန်ရှင်တွေနဲ့ ပုဏ္ဍာတွေ ဘယ်လိုဖြေရှင်းမလဲ ဆက်လက်လေ့လာပါမယ်။ 'ရီကားဆစ်ဖ် စဉ်းစားခြင်း' (thinking recursively) သို့မဟုတ် 'ရီကားဆစ်ဖ်နည်းဖြင့် ပုဏ္ဍာဖြေရှင်းခြင်း' (solving problems recursively) ကို လေ့လာမှာပါ။

ရီကားဆစ်ဖ် စဉ်းစားခြင်း ဥပမာ (၁)

ကွန်နာမှာရှိတဲ့ ဘိပါတွေအားလုံး ကောက်မယ်ဆိုပါစို့။ ဘိပါတစ်ခုနဲ့ အထက်ရှိနိုင်တယ်။ ဘိပါမရှိတာလည်း ဖြစ်နိုင်တယ် ယူဆပါ။ ဒီအတွက် ရီကားဆစ်ဖ်ဖန်ရှင် သတ်မှတ်ပါမယ်။

```
def pick_all_beepers():
    ... # to do soon
```

ဖြေရှင်းမဲ့ကိစ္စတစ်ခုကို ၎င်းကိုယ်တိုင်နဲ့ ပုံပန်းသဏ္ဌာန်တူပြီး အရွယ်အစားအားဖြင့် တစ်ဆင့်ထက် တစ်ဆင့် သေးငယ်တဲ့ ကိစ္စတွေအဖြစ် ခွဲခြမ်းကြည့်ပါတယ်။ ဥပမာ ဘိပါငါးခုရှိတဲ့ ကိစ္စကို ဘိပါလေးခု၊ သုံးခု၊ နှစ်ခု နဲ့ တစ်ခု ရှိတဲ့ ကိစ္စတွေအဖြစ် ခွဲပြီး မြင်ကြည့်ရမှာပါ။ ဒီကိစ္စမှာ ဘိပါအရေအတွက်ဟာ အရွယ်အစားပဲ။ လေးခုရှိတဲ့ ကိစ္စဟာ ငါးခုရှိတဲ့ကိစ္စထက် အရွယ်အားဖြင့် တစ်ဆင့်ငယ်တာပေါ့။ ဘိပါမရှိတာလည်း ဖြစ်နိုင်တော့ သုညဘိပါဟာ အငယ်ဆုံးဖြစ်တယ်လို့ ယူဆနိုင်တယ်။

`pick_all_beepers` ဖန်ရှင်ဟာ လက်ရှိဖြေရှင်းမဲ့ အရွယ်အစားထက် တစ်ဆင့်ငယ်တဲ့ ကိစ္စကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူရပါမယ်။ လက်ရှိဖြေရှင်းမဲ့ ကိစ္စက ဘိပါငါးခု ကောက်ရမယ်ဆိုရင် ဘိပါလေးခု ကောက်နိုင်ပြီးသားလို့ ယူဆရမှာပါ။ n ဘိပါရှိတယ် ဆိုရင် $n-1$ ဘိပါကို ကောက်နိုင်ပြီးသား ယူဆရမယ်။ Top-down နည်းမှာလည်း ဒီလိုပဲ မရှိသေးဘဲ၊ မလုပ်နိုင်သေးဘဲ ရှိတယ်၊ လုပ်နိုင်တယ် မှတ်ယူပြီး စဉ်းစားဖြေရှင်းတာကို ပြန်အမှတ်ရမှာပါ။ ရီကားဆစ်ဖ် စဉ်းစားတဲ့အခါမှာ လက်ရှိသတ်မှတ်နေတဲ့ ဖန်ရှင်ကိုယ်တိုင်ကို ရှိပြီးဖြစ်တယ်လို့ သဘောထားရတာ။

ပြီးတဲ့အခါ လက်ရှိကိစ္စကနေ သူ့ထက်တစ်ဆင့်ငယ်တဲ့ ကိစ္စဖြစ်သွားအောင် ဘာလုပ်ရမလဲ စဉ်းစားရတယ်။ ဘိပါငါးခုကနေ လေးခုဖြစ်အောင် ဘိပါတစ်ခု ကောက်ရမှာပေါ့။ ယေဘုယျပြောရင် n ဘိပါရှိရင် ဆိုရင် $n-1$ ဘိပါဖြစ်သွားအောင် ဘာလုပ်ရမလဲ စဉ်းစားတာ။

လက်ရှိဖန်ရှင်ဟာ $n-1$ ဘိပါကို ကောက်နိုင်ပြီးသားလို့ ယူဆထားတယ်။ n ဘိပါရှိရင် ဘိပါတစ်ခု ကောက်လိုက်ရင် $n-1$ ဘိပါဖြစ်သွားမယ်။ ကျန်နေတဲ့ $n-1$ ဘိပါကောက်ဖို့ လက်ရှိဖန်ရှင်ကိုပဲ ပြန်ခေါ်လိုက်မှာပေါ့။

```
# only partially done
def pick_all_beepers():
    ...
    # to pick (n) beepers
    pick_beeper()
    pick_all_beepers() # assuming it can pick (n - 1) beepers
    ...
```

ရီကားဆစ်ဖ် စဉ်းစားတတ်ဖို့ ဒီအဆင့်က အခရာအကျဆုံးပဲ။ တစ်ဆင့်ငယ်တဲ့ ကိစ္စ $n-1$ ကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူပြီး လက်ရှိကစ္စ n ကို ဘယ်လို ဖြေရှင်းမလဲ စဉ်းစားသွားတာ။

ဖန်ရှင်သတ်မှတ်ချက်က မပြီးသေးပါဘူး။ အသေးငယ်ဆုံး ကိစ္စကို ချွင်းချက်အနေနဲ့ စဉ်းစားရမယ်။ အသေးငယ်ဆုံးကိစ္စက ဘိပါမရှိတာ (သုည) ဖြစ်တယ်။ ဘိပါမရှိရင် ဘာမှလုပ်စရာမလိုဘူး။ n နဲ့ $n-1$ ဘိပါအတွက် အထက်ပါအတိုင်း စဉ်းစားတဲ့အခါ $n \neq 0$ လို့ ယူဆရမှာပါ။ ဒါကြောင့် ဘိပါရှိမှပဲ လုပ်အောင် အခုလိုဖြစ်ရပါမယ်

```
# finished
def pick_all_beepers():
    if beepers_present():
        pick_beeper()
        pick_all_beepers()
```

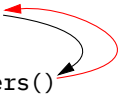
ရီကားဆစ်ဖန်ရှင် မှန်မမှန် စစ်ဆေးခြင်း

ရီကားဆစ်ဖန်ရှင် တက်စ် (test) လုပ်ရင် အသေးဆုံးကိစ္စကနေ စရတယ်။ ပြီးခဲ့တဲ့ ဖန်ရှင်အတွက် အသေးဆုံးက သုညပါ။ ဘိပါမရှိရင် ဖန်ရှင်က မှန်ရဲ့လား အရင်ဆုံး စစ်ကြည့်ပါမယ်။

```
# assume no beeper
pick_all_beepers()
```

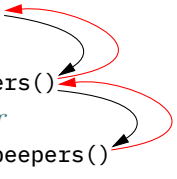
`if` ဘလောက် မလုပ်ဆောင်ဘဲ return ဖြစ်သွားမှာပါ (ရီကားဆစ်ဖ်ကောလ် မဖြစ်လိုက်ဘူး)။ သုညဘီပါအတွက် ဖြစ်သင့်တဲ့အတိုင်း ဖြစ်ပါတယ်။ ဘီပါတစ်ခုပဲ ရှိရင်ရော ဘယ်လိုဖြစ်မလဲ။ ဘီပါကောက်တယ် သုညဘီပါဖြစ်သွားပြီး ရီကားဆစ်ဖ်ကောလ် ဖြစ်မယ်။ တစ်ကြိမ်ပဲ ဖြစ်မယ်။ `if` ဘလောက် အလုပ်မလုပ်ဘဲ return ပြန်မယ်။

```
# initial call
pick_all_beepers()
    # 1st recur
    pick_all_beepers()
```



သုံးခုရှိရင် ရီကားဆစ်ဖ်ကောလ် နှစ်ခါဖြစ်ပြီးမှ return ပြန်မှာပါ။

```
# initial call
pick_all_beepers()
    # 1st recur
    pick_all_beepers()
        # 2nd recur
        pick_all_beepers()
```



မျော်လင့်ထားသလို အလုပ်လုပ်နေပါတယ်။ အထက်ပါအတိုင်း စစ်ကြည့်သွားရင် ဘီပါ သုံး၊ လေး၊ ငါး၊ ... ခု တွေအတွက်လည်း တောက်လျှောက်မှန်နေမှာပါ။ ဘာကြောင့် ပြောနိုင်ရတာလဲ။ အခုလို စဉ်းစားကြည့်နိုင်ပါတယ်။ ရှေ့မှာ စစ်ကြည့်တာ $n = 0, n = 1$ အတွက် မှန်တာ သေချာပြီ။ $n = 2$ အတွက် စစ်မယ်ဆိုပါစို့

```
# start with n = 2
if beepers_present():
    pick_beeper() # after this n = 1
    pick_all_beepers() # works correctly for n = 1
```

$n = 2$ အတွက်မှန်ရင် $n = 3$ အတွက်လည်း ဆက်ပြီး မှန်နေမှာပါ

```
# start with n = 3
if beepers_present():
    pick_beeper() # after this n = 2
    pick_all_beepers() # already works for n = 2
```

$n = 3$ အတွက်မှန်ရင် $n = 4$ အတွက်လည်း မှန်ပြီပေါ့

```
# start with n = 4
if beepers_present():
    pick_beeper() # after this n = 3
    pick_all_beepers() # already works for n = 3
```

ဒီတိုင်းဆက်သွားရင် သုညအပါအဝင် မည်သည့် အပေါင်းကိန်းပြည့် n အတွက်မဆို (non-negative integer) မှန်တယ်ဆိုတာ မြင်နိုင်ပါတယ်။

ရီကားဆစ်ဖ် စဉ်းစားခြင်း ဥပမာ (၂)

ဒုတိယ ဥပမာအနေနဲ့ အခန်း (၃) က အမှိုက်ရှင်းတဲ့ ဥပမာကို ရီကားဆစ်ဖ်နည်းနဲ့ ဖြေရှင်းကြည့်ရအောင်။ လမ်းတစ်လမ်းရှင်းဖို့ ဘယ်လိုစဉ်းစားမလဲ။

> ဖြေရှင်းမဲ့ကိစ္စကို ၎င်းကိုယ်တိုင်နဲ့ သဏ္ဌာန်တူပြီး အရွယ်အစားအားဖြင့် တစ်ဆင့်ထက်တစ်ဆင့် သေးငယ်တဲ့ ကိစ္စတွေအဖြစ် ခွဲခြမ်းကြည့်ရမှာပါ။

လမ်းတစ်ခုရဲ့ အရှည်ကို အရွယ်အစားလို့ ယူဆနိုင်တယ်။ တစ်နည်းအားဖြင့် လမ်းတစ်လျှောက် ကွန်နာအရေအတွက်ဟာ အခုဖြေရှင်းမဲ့ ကိစ္စရဲ့ အရွယ်အစားပဲ။ ကွန်နာတစ်ခုတော့ အနည်းဆုံး ရှိရမယ်။ ဒါကြောင့် အသေးဆုံးက $n = 1$ ဖြစ်တယ်။

> `clean_street` ဖန်ရှင်ဟာ လက်ရှိဖြေရှင်းမဲ့ အရွယ်အစားထက် တစ်ဆင့်ငယ်တဲ့ ကိစ္စကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူရပါမယ်။

အခုကိစ္စအတွက် ကွန်နာငါးခု ရှင်းမယ်ဆိုရင် `clean_street` ဖန်ရှင်ဟာ ကွန်နာလေးခုနဲ့ လမ်းကို ရှင်းနိုင်တယ်လို့ ယူဆရမယ်။ n ကွန်နာပါတဲ့လမ်းအတွက် $n-1$ ကွန်နာကို ရှင်းနိုင်ပြီးသားလို့ ယူဆရမှာဖြစ်ပါတယ်။

> လက်ရှိအရွယ်အစားကို သူထက်တစ်ဆင့်ငယ်တဲ့ ကိစ္စဖြစ်သွားအောင် ဘာလုပ်ရမလဲ စဉ်းစားရတယ်။ ကွန်နာငါးခု လမ်းကိုရှင်းတဲ့အခါ ကွန်နာ လေးခုပဲ ရှင်းစရာကျန်အောင် ဘာလုပ်မလဲ။ ယျေဘုယျပြောရင် n ကွန်နာဆိုရင် $n-1$ ကွန်နာ ရှင်းဖို့လိုတော့အောင် ဘာလုပ်မလဲ။

လမ်းတစ်လမ်းရှင်းတဲ့အခါ လမ်းအစ သို့မဟုတ် လမ်းအဆုံးမှာ အခြားဘက်စွန်းကို မျက်နှာမူထားတယ်။ ရှေ့တစ်ကွန်နာ ရွှေ့လိုက်ရင် ရှင်းစရာ ကွန်နာတစ်ခု လျော့သွားမှာပါ။

> တစ်ဆင့်ငယ်တဲ့ ကိစ္စ $n-1$ ကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူပြီး လက်ရှိကစ္စ n ကို ဘယ်လို ဖြေရှင်းမလဲ စဉ်းစားရပါမယ်။

ကွန်နာငါးခုရှင်းမယ်ဆိုရင် လေးခုကို ရှင်းနိုင်ပြီးသား မှတ်ယူထားရမယ်။ ဘိပါရှိရင်ကောက်၊ ရှေ့တစ်ကွန်နာ ရွှေထားလိုက်ရင် ရှင်းစရာ လေးခုပဲကျန်မယ်။ ဒီလေးခုကို လက်ရှိသတ်မှတ်နေတဲ့ ဖန်ရှင်နဲ့ ရှင်းလိုက်ရုံပဲပေါ့။

```
def clean_street():
    ...
    if beepers_present():
        pick_beeper()
    move()
    clean_street() # assuming already works for n − 1
    ...
```

> အသေးငယ်ဆုံး ကိစ္စကို ချွင်းချက်အနေနဲ့ စဉ်းစားရမယ်။ ကွန်နာတစ်ခုဟာ အသေးငယ်ဆုံးကိစ္စ ဖြစ်တယ်။ ဒီကိစ္စကို ဘယ်လိုဖြေရှင်းမလဲ။

ကွန်နာတစ်ခုပဲ ရှိတယ်ဆိုရင် ဘိပါရှိရင် ကောက်လိုက်ရုံပါပဲ။ n နဲ့ $n-1$ ကွန်နာ အတွက် အထက်ပါအတိုင်း စဉ်းစားတဲ့အခါ $n > 1$ လို့ ယူဆရမှာပါ။ ရှေ့မှာရှင်းနေရင် $n > 1$ မို့လို့၊ မရှင်းတော့ဘူးဆိုရင်

$n = 1$ ဖြစ်နေပြီ။

```
def clean_street():
    if front_is_clear():        # ရှေ့မှာ ကွန်နာတွေ ရှိနေသေးရင်
        if beepers_present():
            pick_beeper()
        move()
        clean_street()
    else:                       # နောက်ဆုံးကွန်နာဆိုရင်
        if beepers_present():
            pick_beeper()
```

အသေးဆုံးကနေစပြီး ဖန်ရှင် အလုပ်လုပ်တာ မှန်/မမှန် စိစစ်ပါ။ ကွန်နာတစ်ခုပဲ ရှိတဲ့လမ်းဆိုရင် စစချင်းပဲ ရှေ့မှာ ပိတ်နေမှာပါ။ `else` အပိုင်း အလုပ်လုပ်မယ်။ ဘိပါရှိရင် ကောက်တယ်။ ကွန်နာနှစ်ခု ရှိတယ်ဆိုရင် စစချင်း ရှေ့မှာ နံရံမရှိဘူး။ `if` အပိုင်း အလုပ်လုပ်မယ်။ ဘိပါရှိရင် ကောက်တယ်၊ ရှေ့တိုးတယ် (နံရံပိတ်သွားပြီ)။ ရီကားဆစ်ဖ်ကောလ် ဖြစ်တယ်။ `else` အပိုင်းလုပ်ပြီးတာနဲ့ `return` စဖြစ်တယ်။

ဒီအတိုင်းဆက်စစ်သွားရင် တစ်ခုထက်ပိုတဲ့ ကွန်နာတွေအတွက်လည်း မှန်အောင် အလုပ်လုပ်နေမယ်ဆိုတာ သက်သေပြနိုင်ပါတယ်။ စာရေးသူ အတွေ့အကြုံအရ အသေးဆုံးနဲ့ သူ့ထက်ကြီးတာ နှစ်ခုသုံးခုလောက်ထိ မှန်တယ်ဆိုရင် နောက်ဟာတွေအတွက် မှားစရာ အကြောင်းမရှိတော့ဘူး။

ရီကားဆစ်ဖ်နည်းနဲ့ လမ်းတစ်လမ်း ရှင်းလို့ရပါပြီ။ ကားရဲလ်ကမ္ဘာတစ်ခုလုံး ရှင်းဖို့ ရီကားဆစ်ဖ် ဆက်ပြီး စဉ်းစားပါမယ်။

- လမ်းအရေအတွက်ဟာ အရွယ်အစားလို့ ယူဆနိုင်တယ်။ အသေးဆုံးက လမ်းတစ်လမ်းပါ။
- ငါးလမ်းရှိရင် ကျန်တဲ့လေးလမ်းကို ရှင်းနိုင်ပြီးသား မှတ်ယူရမယ်။
- (၁) လမ်းရှင်းပြီး အဆုံးမှာ အပေါ်လမ်း ကူးလိုက်ရင် လေးလမ်းပဲကျန်မယ် (လမ်းအရေအတွက် n ရှိရာကနေ $n-1$ ဖြစ်အောင် ဘာလုပ်မလဲ စဉ်းစားတာ)။
- တစ်ဆင့်ငယ်တဲ့ ကိစ္စ $n-1$ ကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူပြီး လက်ရှိကစ္စ n ကို ဘယ်လို ဖြေရှင်းမလဲ စဉ်းစားရပါမယ်။

```
def clean_world():
    ...
    clean_street()
    turn_north()
    change_street()
    clean_world()
    ...
```

- အသေးငယ်ဆုံး ကိစ္စကို ချွင်းချက်အနေနဲ့ စဉ်းစားရမယ်။ လမ်းတစ်လမ်းပဲရှိတာက အသေးငယ်ဆုံး။ လမ်းတစ်လမ်းရှင်းပြီး မြောက်ဘက်လှည့်အပြီး ပိတ်နေပြီဆိုရင်တော့ နောက်ထပ် ဆက်ပြီး ရှင်းစရာ လမ်းမရှိတော့ဘူး။ အပေါ်အဆင့်က လမ်းကူးတာနဲ့ ကျန်တဲ့လမ်းတွေကို ရှင်းတဲ့ကိစ္စကို မြောက်ဘက်လှည့်ပြီးတဲ့အခါ ရှေ့မှာရှင်းနေမှလုပ်ရမှာပါ။

```
def clean_world():
    clean_street()
    turn_north()
    if front_is_clear():
```

```
        change_street()
        clean_world()
```

တစ်လမ်း၊ နှစ်လမ်း၊ သုံးလမ်း အသေးဆုံး ကိစ္စတွေ မှန်/မမှန် စစ်ကြည့်ပါ။ ပရိဂရမ် အစအဆုံး ဖော်ပြပေးထားပါတယ်။ လေ့လာကြည့်ပါ။

```python
# File: clean_world_recur1.py
from stanfordkarel import *

def main():
    clean_world()

def clean_world():
    clean_street()
    turn_north()
    if front_is_clear():
        change_street()
        clean_world()

def clean_street():
    if front_is_clear():        # ရှေ့မှာ ကွန်နာတွေ ရှိနေသေးရင်
        if beepers_present():
            pick_beeper()
        move()
        clean_street()
    else:                       # နောက်ဆုံးကွန်နာဆိုရင်
        if beepers_present():
            pick_beeper()

def change_street():
    move()
    if right_is_blocked():
        turn_left()
    else:
        turn_right()

def turn_right():
    turn_left()
    turn_left()
    turn_left()

def turn_north():
    while not_facing_north():
        turn_left()

if __name__ == "__main__":
```

```
    run_karel_program("clean_world")
```