تقنيات فحص البرمجيات

Software Testing Technologies

مقدمة:

إن أهمية اختبار البرمجيات ومنعكساته على جودتها ليست بحاجة إلى كثير من التأكيد. يقول Deutsch [DEU79] في هذا الصدد:

يشتمل تطوير النظم البرمجية على سلسلة فعاليات إنتاجية تكون فيها فرص دخول الأخطاء البشرية هائلة. قد تبدأ الأخطاء بالحدوث في مستهل الإجرائية حيث تكون الأهداف... قد حُددت تحديداً إما خاطئاً أو غير كامل، وفي مراحل التصميم والتطوير المتأخرة أيضاً .... بسبب عدم قدرة الإنسان على العمل والاتصال بالشكل المطلوب، لذلك يجب أن يرافق تطوير البرمجيات بفعالية ضمان الجودة.

فحص البرمجيات Software Testing هو عملية تنفيذ البرنامج بنية اكتشاف الأخطاء قبل تسليمه للمستخدم. ويتشعب في فئتين أساسيتين:

- **Verification التحقق**: تتم من خلال نمذجة البرمجية المطلوبة باستخدام أدوات نمذجة كشبكات بتري وإجراء عمليات على هذه النماذج قبل الذهاب إلى كتابة الكود. وإذا كان كل شيء صحيح لا يوجد توقف، لا يوجد مجاعة، أداء البرمجيات مناسب .. فإننا نذهب إلى كتابة الكود أو توليد الكود من نماذج البرمجية.
- **Validation إقرار الصلاحية**: تتم نمذجة البرمجية باستخدام أدوات مثل UML وتم كتابة كود البرمجية والآن نريد التحقق من البرنامج المكتوب يعمل بشكل متوافق مع التوصيفات.

من يفحص البرمجيات Tester:

- Developer وهو لطيف في عملية الفحص –يركز على تطوير البرمجيات- ويفهم كيف يعمل النظام ولو كان يعلم بوجود الخطأ لما كتبه.
- Independent Tester يركز على جودة البرمجية وعليه أن يفهم عمل النظام ويركز على اكتشاف العيوب فيه.

كي يكون الاختبار أكثر فعالية، يجب أن يُجرى بواسطة طرف ثالث. نعني باختبار "أكثر فعالية" ذلك الاختبار الذي يتمتع بأعلى احتمال في كشف الأخطاء (الغاية الأولى للاختبار).

يمكن تصنيف فحص البرمجيات إلى أنواع (**حسب كيفية الفحص**):

- ستاتيكي : دون تشغيل البرنامج وإنما من خلال تفتيش الكود عن نماذج معينة من الأخطاء والعيوب.
- ديناميكي: من خلال تشغيل البرنامج ومراقبة خرجه وسلوكه.

يمكن تصنيف فحص البرمجيات إلى أنواع (**حسب متى يتم الفحص**):

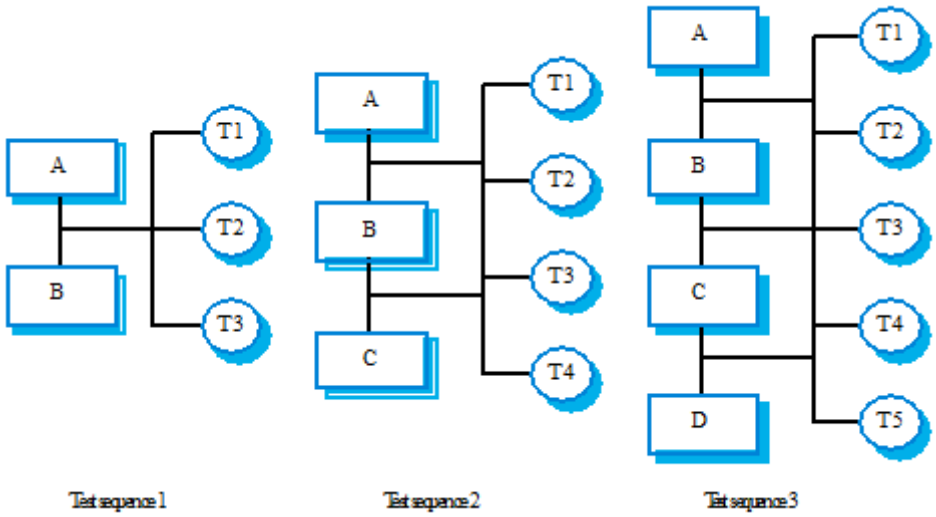
- Unit (Component) test بعد إنهاء كل مكون من مكونات النظام
- Integration test للتأكد من أن المكونات تعمل كما يجب فيما بينها
- Validation test للتأكد من أن النظام يؤدي الوظائف المطلوبة
- System testing للتأكد من أن النظام يؤدي الوظائف المطلوبة وبالجودة المطلوبة

يجب أن تجرى الاختبارات "للقطع الصغيرة" في البداية، ثم ينتقل تدريجياً بها لتشمل "القطع الكبيرة". عموماً، تركّز الاختبارات الأولى المخطط لها والمنفذة على مجتزآت البرنامج المستقلة، ومع تقدم الاختبار يتركّز الاهتمام في محاولة إيجاد الأخطاء في مجموعات المجتزآت المتكاملة، ثم في النظام ككل في نهاية المطاف

الاختبار الجيد:

- يتميز باحتمال عال لإيجاد الخطأ.
- لايحوي تكرار

اختبار التكامل التزايدي:



يتم أولاً اختبار تكامل المكونين ألأهم في البرمجية A وB بثلاثة حالات اختبار (T1, T2, T3) –اصبر قليلاً وسأشرح ما تعنيه حالة الاختبار-.

ومن ثم يتم اختبار تكامل المكونات التالية في البرمجية A وB و C بأربعة حالات اختبار (T1, T2, T3, T4). تم إعادة حالات الاختبار T1, T2, T3 بالرغم من أن T4 هي التي تخص مكاملة المكون C مع المجموعة السابقة. وذلك لربما إن إضافة المكون C أثرت على ترابط A وB فيجب التأكد من الاختبارات T1, T2, T3 مازالت صحيحة

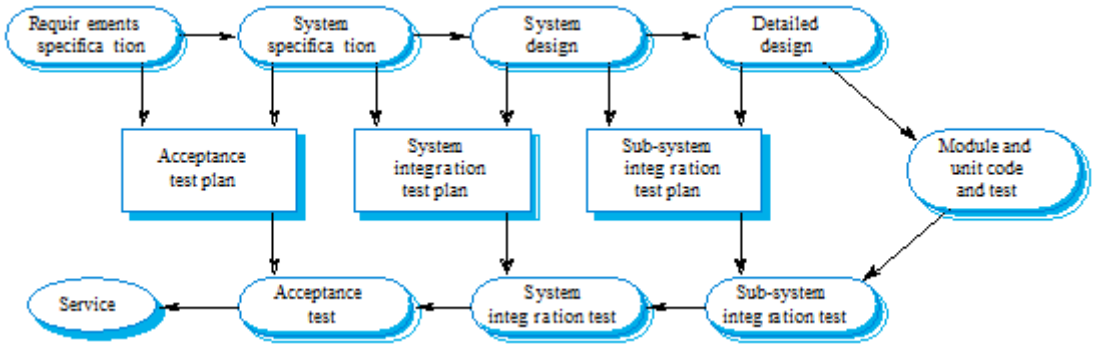
ومن ثم يتم اختبار تكامل المكونات التالية في البرمجية A وB و C و D بخمس حالات اختبار (T1, T2, T3, T4, T5). تم إعادة حالات الاختبار T1, T2, T3, T4 بالرغم من أن T5 هي التي تخص مكاملة المكون D مع المجموعة السابقة. وذلك لربما إن إضافة المكون D أثرت على ترابط A وB و Cفيجب التأكد من الاختبارات T1, T2, T3, T4 مازالت صحيحة.

وهكذا .....

نموذج التطوير V:

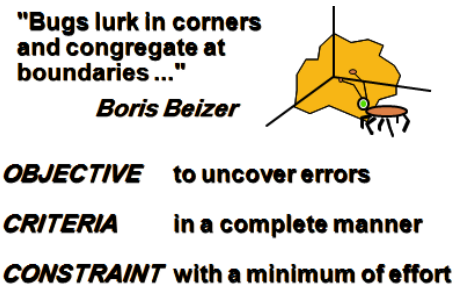
نلاحظ أننا عندما:

- وصفنا متطلبات النظام .. نستطيع توصيف Acceptance Test (صحيح هو آخر ما نعمله ولكن أول ما نوصفه)
- وصفنا بنية النظام .. نستطيع توصيف Integration Test (بالرغم من أننا لا نستطيع القيام به قبل تطوير مكونات البرمجية)
- وصفنا عمل مكونات النظام .. نستطيع توصيف Unit Test (صحيح هو آخر ما نوصفه ولكن أول ما نقوم نعمله)



تصميم حالات الاختبار Test Case Design (---- الموضوع الأهم في فحص البرمجيات ----)

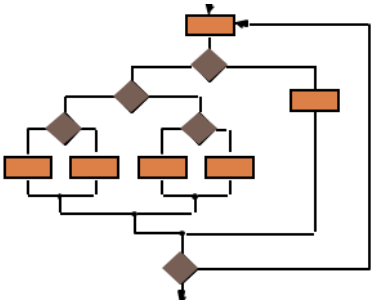
سنتعلم كيف نصمم حالات الاختبار



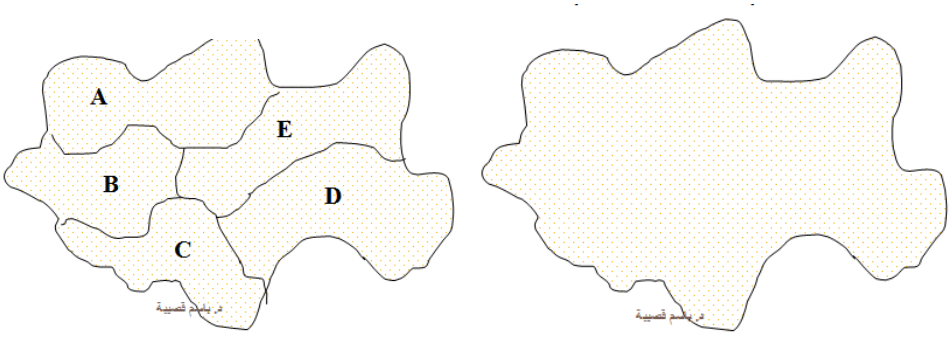
تاريخ: عندما تعطل أول حاسب (وكان عملاقا) جاء المهندسون لعند كبيرهم وسألهم ما الخطب فقالوا Bug أي أن حشرة كانت سبب التماس الذي عطل الحاسب وأصبحت مثلاً على سبيل الطرفة فعندما لا يعمل برنامج ما نقول Bug.

القدر: أن الحشرات تتجمع في زوايا المنزل وتمشي على الفاصل (الحدود) بين الأرض والجدار وكذلك أخطاء البرامج هي في المناطق الحدية (مثلاً .. عند نهاية مصفوفة : تعمل for ولكنها تتجاوز بعد المصفوفة بواحد) وهذا ما نسميه بالعيب Fault ولكن ليس من الضرورة أن يظهر فورا فإن كنت تعبر المصفوفة باحثاً عن عنصر موجود فيها فلن تصل إلى النهاية ولن ترى العيب .. وإن لم يكن العنصر غير موجود فستصل إلى ما بعد نهاية المصفوفة وينتج خطأ Error و لو أوقف هذا الخطأ البرنامج لأصبح إخفاق Failure

لذلك لاكتشاف الأخطاء نحتاج إلى تصميم حالات اختبار .. كل حالة اختبار هي مجموعة دخل للبرنامج (مثل في برنامج البحث عن عنصر من مصفوفة ... حالة الفحص = البحث عن عنصر غير موجود في المصفوفة .. للتأكد من أن حلقة عبور المصفوفة لا تتجاوز حدود المصفوفة)



إن أي برنامج يقبل عدد لا نهائي من تشكيلات الدخل (برنامج جمع عددين ..2+3 .. 8+5 ..22+44 .. إلخ) فمن المستحيل اختبار كل تشكيلة دخل ممكنة وبالتالي من المستحيل إختبار كل التشكيلات الممكنة.. بل أخذ تقسيم كل تشكيلات الدخل الممكنة إلى صفوف تكافؤ بحيث أي اختبار لنقطة (تشكيلة دخل) من هذا الصف التكافؤ كما لو أنك اختبرت كل نقاطه ..

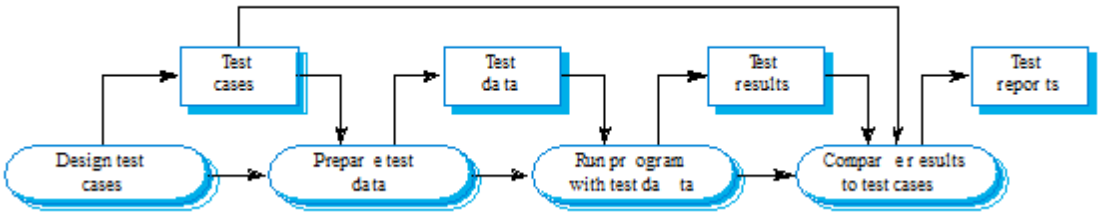


كيف نقسم فضاء تشكيلات الدخل (حالات الفحص) إلى صفوف تكافؤ من حالات الفحص (تشكيلات الدخل) بالحفاظ على معايير تصميم حالات الفحص:

- اكتشاف الأخطاء
- بشكل كامل (دون أن ننسى اكتشاف أحدها)
- بأقل جهد ووقت ممكن (بأقل عدد ممكن من حالات الفحص)

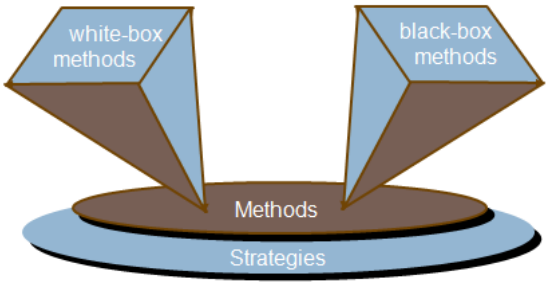
أي اختبار العدد الأصغري من حالات الفحص واكتشاف كل الأخطاء

يبين الشكل التالي إجرائية تصميم حالات الاختبار واختبار النظام (واضحة)



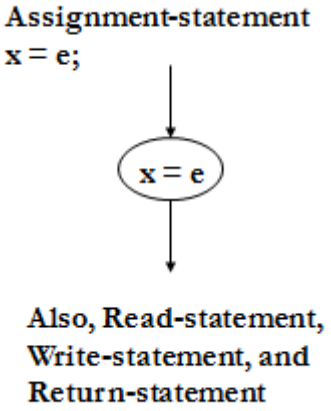
لدينا منهجيتين لتصميم حالات الفحص:

- White Box: باختصار سنعتمد على الكود لاستنتاج حالات الفحص.
- Black Box: باختصار سنعتمد على توصيف وظائف النظام لاستنتاج حالات الفحص.

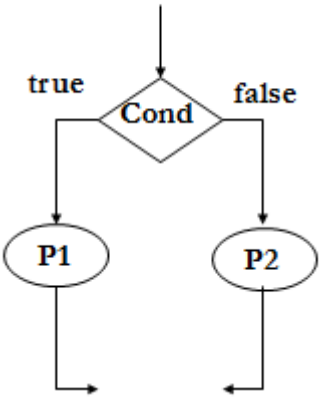


منهجية White BOX

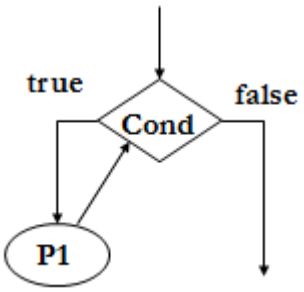
يجب تحويل البرنامج إلى Control Flow Graph من خلال تحويل تعليمات البرنامج إلى نماذج Flow Charts وهي التالية:



if (cond) then P1 else P2

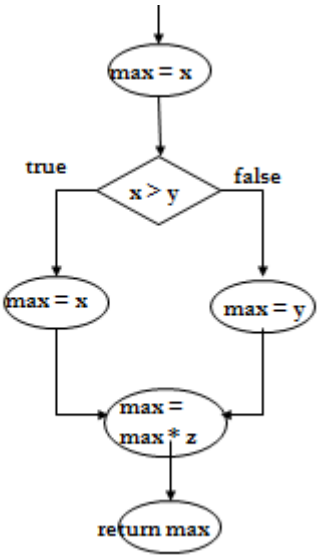


while (cond) P1



مثال 1 .. نحول البرنامج إلى Control Flow Graph

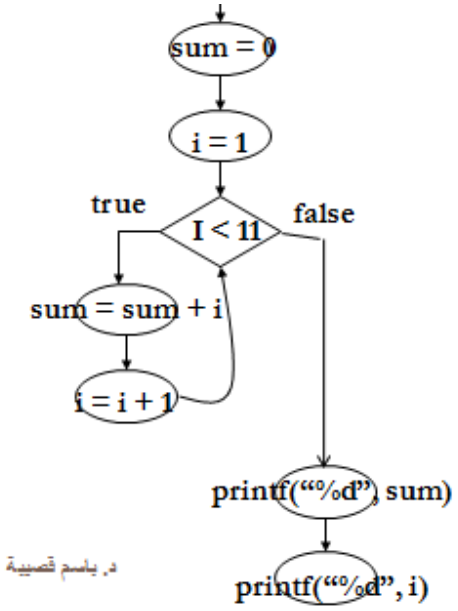
```
int main(int x, y, z) {
      int max;
      max = x
      if (x > y)
            max = x;
      else
            max = y;
      max = max * z;
      return max;
}
```



مثال 2 .. نحول البرنامج التالي إلى Control Flow Graph

```
int main() {
    int sum, I;

    sum = 0;
    i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d",sum);
    printf("%d",i);
}
```



Statement Coverage (لإيجاد صفوف التكافؤ) وتعني يجب اختيار حالات فحص بحيث يتم تغطية كل تعليمات البرنامج على الأقل مرة واحدة (تنفيذ كل تعليمات البرنامج على الأقل مرة واحدة).

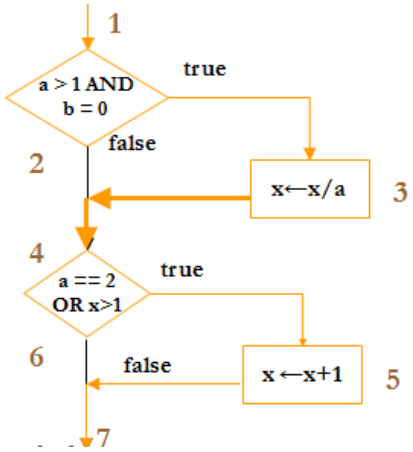
بفرض لدينا مخطط التحكم لبرنامج بالشكل التالي .. يوجد مسار واحد يكفي للمرور على تعليمتي البرنامج عند النقطة 3 والنقطة 5... حالة الفحص هي الدخل الذي يجعل البرنامج يسير بالمسار المطلوب

The following path is sufficient for statement coverage:

1 – 3 – 4 – 5 – 7

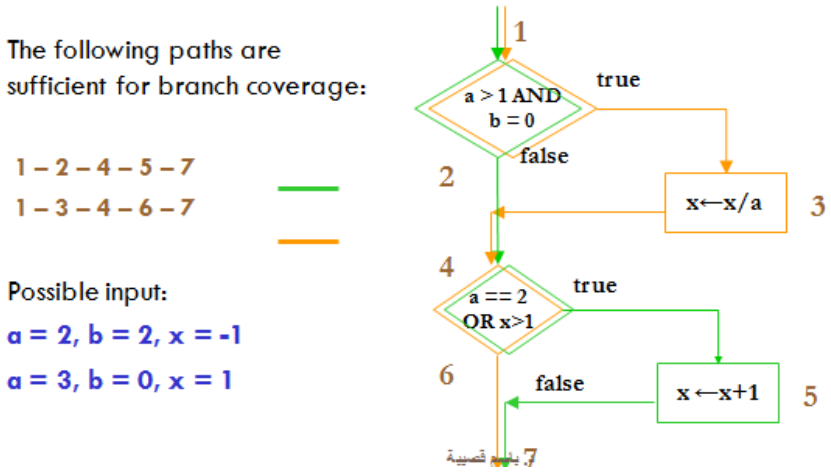
Possible input:

a = 2, b = 0, x = 4



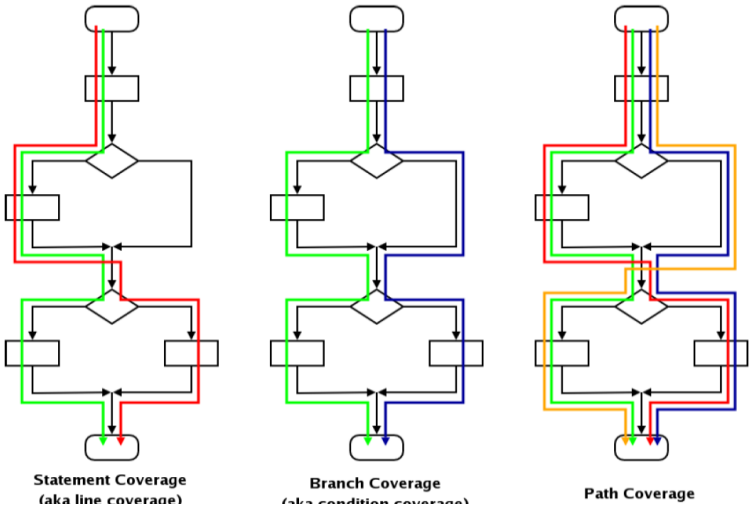
إن Statement Coverage ضعيفة ولذلك تم استخدام **Branch Coverage** (لإيجاد صفوف التكافؤ) وتعني يجب اختيار حالات فحص بحيث يتم تغطية كل فروع الشروط في البرنامج على الأقل لمرة واحدة (السير من فرع True و ال False لكل اختبار شرط في البرنامج).

بفرض لدينا مخطط التحكم لنفس البرنامج السابق بالشكل التالي .. يوجد مسارين كافيين للمرور على كل فروع اختبارات البرنامج .. كما يبين حالتي الفحص المناسبتين للمسارين كي يسير البرنامج كما يراد له



إن Branch Coverage غير كافية ولذلك تم استخدام **Path Coverage** (لإيجاد صفوف التكافؤ) وتعني يجب اختيار حالات فحص بحيث يتم تغطية كل المسارات التي يمكن للبرنامج السير فيها على الأقل لمرة واحدة.

يوضح الشكل التالي الفروق بين أساليب التغطية لإنتاج صف تكافؤ (إن كل مسار يعوض عن صف تكافؤ)

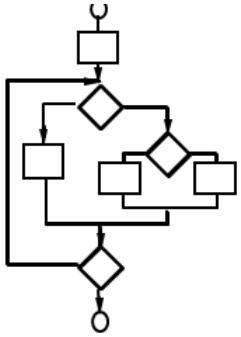


إن Path Coverage كاف في منهجية White Box

طريقة إيجاد Path Coverage:

1 - نحسب عدد المسارات = عدد الحجرات المغلقة + 1 = عدد الشروط في البرنامج + 1

التعقيد المساراتي *cyclomatic complexity* هو مقياس برمجي يعطي قياساً كمياً للتعقيد المنطقي لبرنامج ما. عند استخدام هذا المقياس في سياق طريقة اختبار المسار الأساسي، تحدد القيمة المحسوبة للتعقيد المساراتي عدد *المسارات المستقلة* في *المجموعة الأساسية* *basis set* للبرنامج، وتعطي الحد الأعلى لعدد الاختبارات الواجب إجراؤها لكي نضمن تنفيذ جميع التعليمات مرة على الأقل.



First, we compute the cyclomatic complexity:

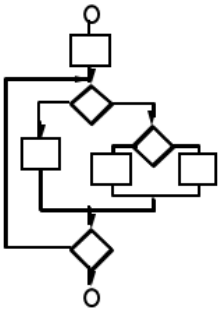
number of simple decisions + 1

or

number of enclosed areas + 1

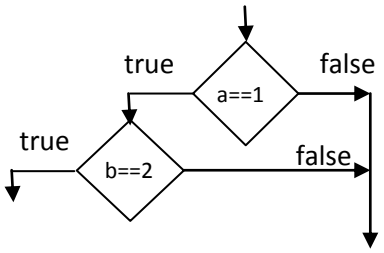
In this case, V(G) = 4

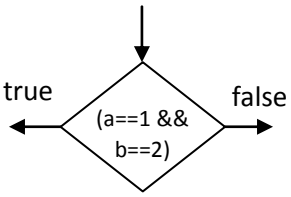
ملاحظة: يجب أن تكون الشروط بسيطة:



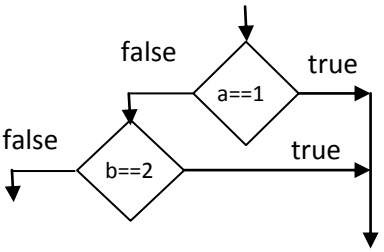
- you don't need a flow chart, but the picture will help when you trace program paths
- count each simple logical test, compound tests count as 2 or more
- basis path testing should be applied to critical modules

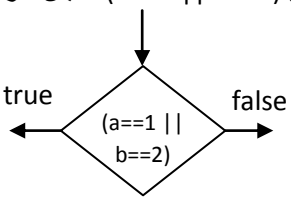
يعني if(a==1 && b==2) ليس شرطا بسيطا





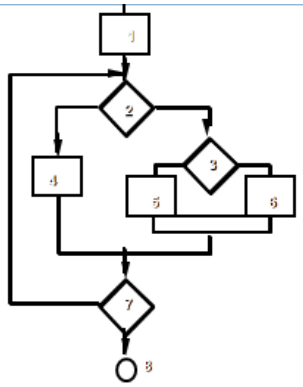
يعني if(a==1 || b==2) ليس شرطا بسيطا





2 - نحدد المسارات المستقلة

المسار المستقل هو كل مسار في البرنامج يؤدي إلى إدخال مجموعة جديدة من عبارات المعالجة أو إدخال شرط جديد. عندما يعبّر عن مسار مستقل بواسطة بيان التدفق يجب أن يتقدم المسار حرفاً واحداً على الأقل من الأحرف التي لم تُعبّر قبل تعريف المسار.



Next, we derive the independent paths:

Since V(G) = 4, there are four paths

Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2,4,...7,8

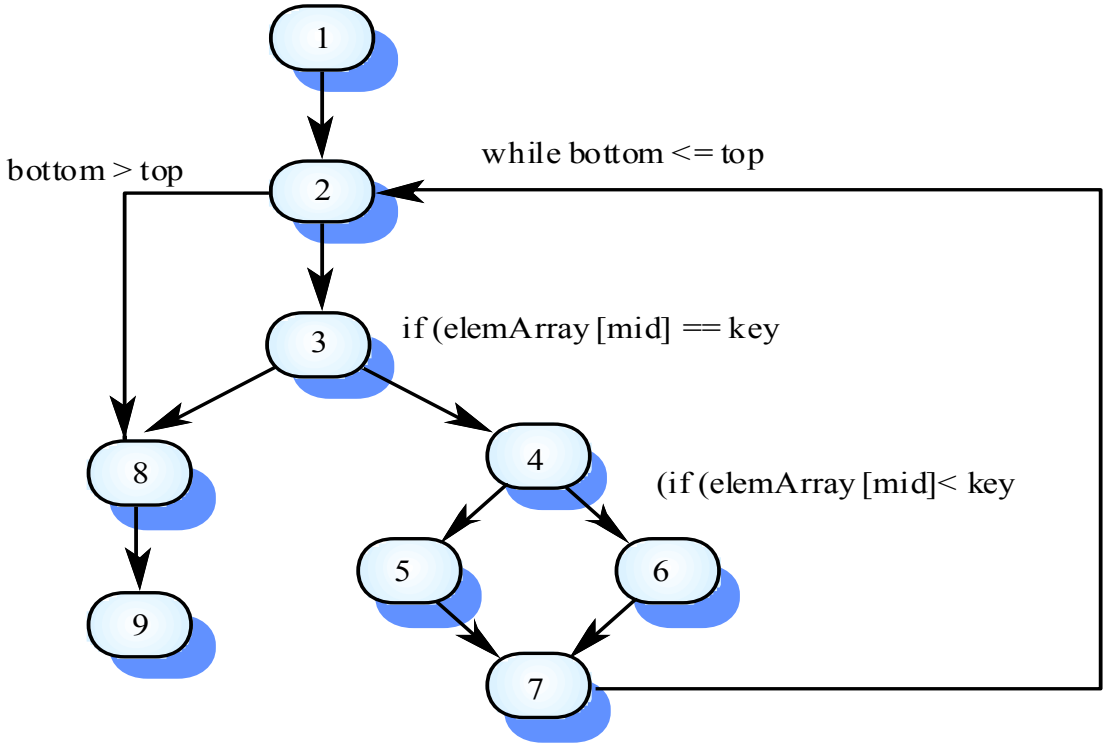
Finally, we derive test cases to exercise these paths.

مثال: برنامج البحث الثنائي

```
class BinSearch {

    public static void search ( int key, int [] elemArray, Result r )
    {
        int bottom = 0 ;
        int top = elemArray.length - 1 ;
        int mid ;
        r.found = false ; r.index = -1 ;
        while ( bottom <= top )
        {
            mid = (top + bottom) / 2 ;
            if (elemArray [mid] == key)
            {
                r.index = mid ;
                r.found = true ;
                return ;
            } // if part
            else
            {
                if (elemArray [mid] < key)
                    bottom = mid + 1 ;
                else
                    top = mid - 1 ;
            }
        } //while loop
    } // search
} //BinSearch
```

نرسم Control Flow Graph



لدينا 3 حجرات فارغة، فلدينا أربعة مسارات:

المسار 1-2-8-9 : حالة الفحص المكافئة (المصفوفة فارغة)

المسار 1-2-3-8-9 : حالة الفحص المكافئة (العنصر الذي نبحث عنه في منتصف المسافة تماما)

المسار 1-2-3-4-5-7-8-9 : حالة الفحص المكافئة (العنصر الذي نبحث عنه في منتصف النصف العلوي تماما)

المسار 1-2-3-4-6-7-8-9 : حالة الفحص المكافئة (العنصر الذي نبحث عنه في منتصف النصف السفلي تماما)