

Lecture 2: Software Metrics

Độ đo phần mềm

IT4501

Bùi Thị Mai Anh (anhbtm@soict.hust.edu.vn)

Bộ môn Công nghệ phần mềm

Viện công nghệ thông tin và truyền thông

1

Nội dung bài học

2

- Mục đích của việc sử dụng các độ đo phần mềm
- Các độ đo trực tiếp
- Độ đo của Halstead
- Chỉ số phức tạp McCabe
- Chỉ số bảo trì MI
- Các độ đo hướng đối tượng

2

Làm thế nào để đảm bảo dự án phần mềm thành công?

3

Để đảm bảo được một dự án được thực hiện thành công chúng ta cần phải hiểu được:

- Phạm vi công việc cần được thực hiện
- Các rủi ro có thể có trong dự án
- Các nguồn tài nguyên cần thiết
- Các tác vụ công việc cần được hoàn thành
- Các cột mốc cần được theo dõi
- Chi phí cần thiết
- Kế hoạch cần phải tuân theo

3

Hoạt động quản lý dự án phần mềm

4

- Trước khi một dự án được lên kế hoạch
 - Các mục tiêu và phạm vi của dự án cần được đưa ra
 - Các giải pháp thay thế cần được xem xét
 - Các ràng buộc về kỹ thuật và quản lý cần được xác định
- Thông tin này cần được yêu cầu để đánh giá được các chi phí, công việc và kế hoạch cụ thể để thực hiện

4

Độ đo phần mềm

5

- Các độ đo giúp chúng ta hiểu được quy trình kỹ thuật được sử dụng khi phát triển một sản phẩm.
 - Quy trình cần được đo lường để có thể cải tiến và sản phẩm cũng cần được đo lường để tăng chất lượng
- Đo lường các dự án phần mềm vẫn còn đang tranh cãi vì:
 - Cách đo lường phần mềm vẫn chưa rõ ràng và nó phụ thuộc cụ thể vào từng dự án
 - Các độ đo cũng chưa rõ ràng, phụ thuộc vào con người, quy trình và sản phẩm cần được so sánh sử dụng những độ đo đó

Tại sao cần sử dụng các độ đo?

6

- Không có các độ đo thì sẽ không có cách nào để xác định xem một quy trình có cần được cải tiến hay không
- Các độ đo cho phép thiết lập các mục tiêu có ý nghĩa để cải tiến chất lượng sản phẩm
- Các độ đo cho phép một tổ chức xác định được các lí do của những lỗi có ảnh hưởng nhất đến quá trình phát triển phần mềm

Áp dụng các độ đo

7

Khi các độ đo được áp dụng vào một sản phẩm chúng cho phép xác định:

- Những yêu cầu nào của người dùng có khả năng sẽ bị thay đổi hoặc cần được thay đổi
- Những mô đun nào có khả năng lỗi nhiều nhất
- Những cách kiểm thử như thế nào cần được lên kế hoạch cho từng mô-đun

7

Phân loại độ đo phần mềm

8

Các độ đo của phần mềm thì thường được phân vào hai loại:

- các độ đo trực tiếp
- các độ đo gián tiếp
- Các độ đo trực tiếp thông thường bắt nguồn từ một tính năng hoặc một đặc tính cụ thể ví dụ như chiều dài code
- Các độ đo trực tiếp hay dùng để đánh giá mã nguồn gồm: dòng code, tốc độ thực thi, kích thước bộ nhớ yêu cầu, thông báo lỗi

8

Các độ đo gián tiếp

9

- Các độ đo gián tiếp liên kết một độ đo với một mục tiêu đo đặc cụ thể như chất lượng dựa trên các thất bại đếm được của dự án.
- Thông thường các mục tiêu cụ thể có thể là:
 - chức năng,
 - chất lượng,
 - độ phức tạp,
 - tính hiệu quả,
 - độ tin cậy,
 - khả năng bảo trì được

9

Code Complexity Measurements Các độ đo độ phức tạp mã nguồn

10

- Độ phức tạp mã nguồn có tương tác trực tiếp đến tỉ lệ lỗi và độ mạnh của chương trình ứng dụng.
- Các công cụ để đo đặc độ phức tạp mã nguồn:
 - Testwell CMT++ để đánh giá độ phức tạp mã nguồn C/C++ hoặc C#
 - Testwell CMTJava cho mã nguồn Java
- Mã nguồn có độ phức tạp tốt thì sẽ chứa ít lỗi, dễ hiểu và dễ bảo trì hơn những mã nguồn có độ phức tạp lớn

10

Làm thế nào để sử dụng độ phức tạp mã nguồn?

11

- Các độ đo về độ phức tạp mã nguồn thường được sử dụng để định vị những khu vực mã nguồn phức tạp
 - Bài toán định vị lỗi phần mềm dựa trên các kỹ thuật NLP (bug localization)
 - Bài toán xác định nguyên nhân gây lỗi dựa trên cấu trúc mã nguồn (fault localization)
- Để đạt được phần mềm với chất lượng tốt và giảm thiểu chi phí kiểm thử và bảo trì thì độ phức tạp mã nguồn cần được ước tính càng sớm càng tốt trong quá trình viết mã
 → người phát triển cần sửa chữa lại code của mình khi những giá trị khuyên dùng về độ phức tạp mã nguồn của mình bị vượt quá.

11

Code complexity metrics Các độ đo phức tạp

12

- Metrics được sử dụng bởi Testwell CMT++/CMTJava
 - Lines of code metrics
 - McCabe Cyclomatic number
 - Halstead Metrics
 - Maintainability Index

12

Lines of code metrics Số dòng code

13

- Là độ đo cơ bản nhất để tính toán độ phức tạp của mã nguồn
- Đơn giản, dễ tính toán và dễ hiểu
- Có nghịch lý gì với độ đo này hay không?
 - Cùng 1 thuật toán, mã nguồn ngắn hơn thì sẽ ít phức tạp hơn mã nguồn dài?
 - Chất lượng mã nguồn? Tính dễ đọc, dễ hiểu, dễ bảo trì và kiểm thử?

13

Lines of code metrics Số dòng code

14

Testwell CMT++/CMTJava calculates the following lines of code metrics

- LOCphy: number of physical lines
- LOCblk: number of blank lines (a blank line inside a comment block is considered to be a comment line)
- LOCpro: number of program lines (declarations, definitions, directives and code)
- LOCcom: number of comment lines

14

Lines of code metrics - Recommendations Những giá trị khuyên dùng cho chiều dài mã nguồn

15

- **Function length:** chiều dài của 1 hàm/phương thức
 - Chiều dài hàm cần trong khoảng từ 4 đến 40 dòng code
 - Mỗi định nghĩa hàm thì chứa ít nhất một khuôn mẫu hàm, 1 dòng code, 1 cặp ngoặc nhọn xác định thân hàm → như vậy là cần tối thiểu 4 dòng
 - Một hàm có chiều dài lớn hơn 40 dòng thì có nhiều khả năng là nó là gộp của hơn hai hàm và có thể tách được thành các hàm nhỏ hơn để giảm thiểu chiều dài của hàm
 - Ngoại lệ: hàm chứa 1 câu lệnh lựa chọn (ví dụ câu lệnh switch – case trong C/C++) với nhiều nhánh lựa chọn (ví dụ 100 nhánh sẽ làm thân hàm có thể hơn 104 dòng code).
- Chất lượng mã nguồn sẽ ảnh hưởng như thế nào khi hàm được tách thành các hàm nhỏ hơn?

15

Lines of code metrics - Recommendations Những giá trị khuyên dùng cho chiều dài mã nguồn

16

- **File length:** chiều dài của 1 file mã nguồn (.c, .cpp, .java...)
 - Một file nên có chiều dài trong khoảng từ 4 đến 400 dòng code tức là khoảng 10 hàm
 - Phần tử nhỏ nhất của một file mã nguồn có thể là 1 hàm và chiều dài tối thiểu của nó do đó là 4 dòng code
 - Những file mã nguồn có chiều dài lớn hơn 400 dòng code thì quá dài để hiểu được toàn bộ ý nghĩa của file đó.
 - Do đó nên được tách ra thành các file mã nguồn khác nhau (ví dụ nhóm các hàm lại thành các nhóm chức năng khác nhau)

16

Lines of code metrics - Recommendations Những giá trị khuyên dùng cho chiều dài mã nguồn

17

• Comments: các chú thích trong mã nguồn

- ít nhất từ **30%** đến **75%** dòng của một file mã nguồn nên là các chú giải để đảm bảo tính dễ hiểu của mã nguồn
- Nếu ít hơn **1/3** của một file là comment thì hoặc là file mã nguồn đó cực kỳ tầm thường hoặc là file mã nguồn đó không được giải thích tốt.
- Nếu nhiều hơn **¾** của một file là comment thì file đó không phải là mã nguồn mà là một tài liệu đặc tả ☺
- Ngoại lệ: Trong một file đặc tả về định dạng file ví dụ headers của một file .mp3, tỉ lệ % comment có thể vượt quá **75%**.

17

Halstead's Metrics Các độ đo của Halstead

18

- Đề xuất bởi Maurice Halstead
- Ra đời vào năm 1977
- Được sử dụng trong thực tế rất nhiều kể từ khi ra đời
- Một trong những độ đo cơ bản nhất về độ phức tạp của mã nguồn
- Là 1 chỉ số rất mạnh để đánh giá độ phức tạp
- Thường được sử dụng như 1 phép đo trong bảo trì phần mềm

18

Halstead's Metrics (cont.)

19

- Các độ đo Halstead thì được tính toán dựa trên việc phân tích mã nguồn thành các token (các thẻ) và phân loại các thẻ đó thành hai loại:
 - toán tử - operator
 - toán hạng - operand
- Chúng ta sẽ tính toán các giá trị độ đo dựa trên các thẻ đã phân loại đó bằng cách tính
 - Số các toán tử phân biệt kí hiệu là n_1
 - Số các toán hạng phân biệt kí hiệu là n_2
 - Tổng số các toán tử kí hiệu là N_1
 - Tổng số các toán hạng kí hiệu là N_2
- Tất cả các số đo khác của Halstead thì đều được tính toán dựa trên 4 giá trị cơ bản này và sẽ được mô tả thông qua các công thức trong phần tiếp theo

19

Operators and Operands

20

- Operators – Toán tử
 - traditional: +, -, *, /, ++, --, etc.
 - keywords: return, if, continue, break, try, catch etc.
- Operands – Toán hạng
 - Identifiers: tên biến, tên hàm, tên file...
 - constants

20

Halstead's Metrics

Một số độ đo của Halstead

21

- **Program length (N)**: chiều dài của chương trình được tính bằng tổng số toán tử và toán hạng có trong chương trình:
 - $N = N_1 + N_2$
- **Vocabulary size (n)**: kích thước của tập từ vựng được tính bằng tổng số các toán tử và toán hạng phân biệt
 - $n = n_1 + n_2$
- **Program volume (V)**: kích thước của chương trình
 - $V = N * \log_2(n)$
- **Kích thước chương trình** miêu tả kích thước cài đặt của một chương trình hay thuật toán

21

Halstead's metrics - Recommendations

Một số khuyến cáo về giá trị độ đo Halstead

22

- Kích thước của một hàm nên có giá trị volume halstead từ 20 đến 1000.
 - Một hàm chỉ có một dòng và không có tham số thì vào khoảng 20.
 - Một hàm có kích thước volume hơn 1000 thì về cơ bản có thể làm quá nhiều việc trong hàm và chúng ta có thể nên tách hàm đó thành các hàm nhỏ
 - Kích thước của một file mã nguồn nên từ 100 đến 8000
 - Giá trị volume đo được của một file mã nguồn $V(G)$ và chỉ số LOCpro nên nằm trong khoảng giới hạn quy định của nó

22

Other Halstead's metrics Các chỉ số khác của Halstead

23

- **Difficulty level (D): độ khó của chương trình**
 - Được coi như là mức độ tiềm ẩn lỗi của chương trình thì tỉ lệ với số các toán tử phân biệt của chương trình
 - D cũng được coi như tỉ lệ với số giữa tổng số toán hạng và số toán hạng phân biệt của chương trình
 - Ví dụ như, cùng một toán hạng những được sử dụng quá nhiều lần trong chương trình thì khả năng tiềm ẩn lỗi sẽ cao hơn và đó cần được sửa đổi mã nguồn để làm giảm số lần cùng một toán hạng được sử dụng trong chương trình.
 - Công thức tính D là $D = (n_1/2) * (N_2/n_2)$
- **Program level (L): mức độ của chương trình**
 - Mức độ của chương trình là số đo có giá trị là đảo ngược của mức độ tiềm ẩn lỗi D
 - Ví dụ, một chương trình ở mức độ thấp thì sẽ nhiều khả năng tiềm ẩn lỗi hơn là một chương trình ở mức độ cao
 - $L = 1/D$

23

Other Halstead's metrics

24

- **Effort to implement (E): công sức cài đặt chương trình**
 - Công sức cài đặt chương trình (E) là công sức phải bỏ ra để cài đặt mã nguồn chương trình hoặc để đọc hiểu mã nguồn chương trình
 - Độ đo này tỉ lệ với kích thước và độ khó của chương trình
 - $E = V * D$
- **Time to implement (T): thời gian cài đặt chương trình**
 - Thời gian cần thiết để cài đặt hoặc để hiểu được mã nguồn của chương trình
 - Halstead đã tìm ra rằng chia công sức cài đặt E cho 18 sẽ cho một giá trị xấp xỉ về thời gian cài đặt tính theo giây
 - $T = E/18$

24

Other Halstead's metrics

25

- **Number of delivered bugs (B): số lượng lỗi khi bàn giao**

- Chỉ số này tương quan với độ phức tạp chung của phần mềm và được tính bằng công thức $B=(E^2/3)/3000$.
- Chỉ số này ước tính số lỗi có thể có của chương trình
- Số lượng lỗi của một file mã nguồn nền nhỏ hơn 2.
- Các kinh nghiệm thực tế đã chỉ ra rằng khi lập trình với C hoặc C++, một file mã nguồn luôn luôn có nhiều lỗi tiềm ẩn hơn chỉ số B gợi ý của mã nguồn
- B là một trong số những chỉ số rất quan trọng cho kiểm thử tự động. Số lượng lỗi bàn giao cũng xác xỉ số lượng lỗi thực tế của một module.
- Mục đích khi kiểm thử (nhất là kiểm thử tự động) là phải tìm được ít nhất B lỗi trong mã nguồn.

25

Halstead's metric Example Ví dụ về độ đo Halstead

- Cho 1 đoạn mã nguồn viết bằng C
- Hãy tính các chỉ số cơ bản của Halstead
 - n_1
 - n_2
 - N_1
 - N_2

```
void sort ( int *a, int n ) {
    int i, j, t;

    if ( n < 2 ) return;
    for ( i=0 ; i < n-1; i++ ) {
        for ( j=i+1 ; j < n ; j++ ) {
            if ( a[i] > a[j] ) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}
```

$V = 80 \log_2(24) \approx 392$

26

Halstead Example (cont.)

- Bước 1:
 - Liệt kê tất cả các toán tử và toán hạng trong mã nguồn
 - Đếm số lần xuất hiện của từng toán tử và toán hạng
- Bước 2:
 - Tính toán V, D, E...

```
void sort ( int *a, int n ) {
    int i, j, t;

    if ( n < 2 ) return;
    for ( i=0 ; i < n-1; i++ ) {
        for ( j=i+1 ; j < n ; j++ ) {
            if ( a[i] > a[j] ) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}
```

$V = 80 \log_2(24) \approx 392$

- Ignore the function definition
- Count operators and operands

3 <	3 {	1 0
5 =	3 }	2 1
1 >	1 +	1 2
1 -	2 ++	6 a
2 ,	2 for	8 i
9 :	2 if	7 j
4 (1 int	3 n
4)	1 return	3 t
6 []		

	Total	Unique
Operators	N1 = 50	n1 = 17
Operands	N2 = 30	n2 = 7

McCabe Cyclomatic Number Chỉ số phức tạp của McCabe

28

- Chỉ số phức tạp $v(G)$ được đưa ra bởi Thomas McCabe năm 1976
- Là các số đo được tính toán dựa trên các đường độc lập tuyến tính trong một chương trình.
- Chỉ số phức tạp McCabe được tính toán dựa trên đồ thị luồng điều khiển của chương trình.
- Đây là một trong số các số đo được sử dụng rộng rãi trong công nghiệp phần mềm để tính toán được độ phức tạp của chương trình/mã nguồn.

McCabe Cyclomatic Number – $v(G)$

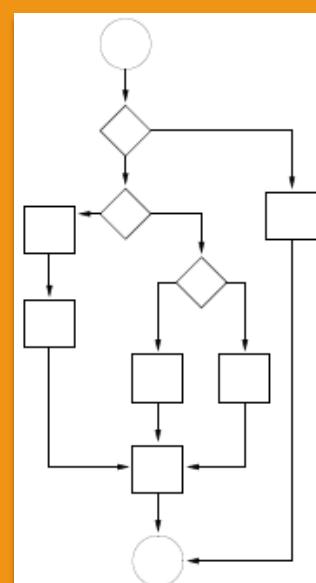
29

- $v(G)$ là số lượng các nhánh trong đồ thị luồng điều khiển của chương trình
- $v(G) = 1$ khi chương trình chỉ gồm một câu lệnh
- Đối với một hàm đơn, $v(G)$ sẽ luôn nhỏ hơn hoặc bằng các điểm rẽ nhánh trong chương trình
- Chỉ số phức tạp của chương trình càng lớn, càng nhiều nhánh cần thực thi trong chương trình và độ khó của chương trình sẽ càng cao

29

McCabe Cyclomatic Number Công thức tính chỉ số McCabe

- Thể hiện cấu trúc của chương trình:
 - Control flow: luồng điều khiển
 - Data flow: luồng dữ liệu
- Các chỉ số của đồ thị:
 - Number of vertices: số lượng đỉnh
 - Number of edges: số lượng cạnh
 - Độ sâu
- $v(G) = \#edges - \#vertices + 2$
- Đối với sơ đồ luồng điều khiển:
 - $v(G) = \#\text{binaryDecision} + 1$
 - $v(G) = \#\text{IFs} + \#\text{LOOPS} + 1$



30

McCabe Cyclomatic Number

Ý nghĩa của chỉ số McCabe

31

- Đối với kiểm thử tự động, chỉ số McCabe là một chỉ số rất quan trọng để ước lượng số các trường hợp kiểm thử cần thiết đối với một function/module
- Độ phức tạp McCabe biểu diễn độ phức tạp luồng điều khiển chương trình
 - Chúng ta có thể rõ ràng thấy rằng các module/hàm có chỉ số McCabe cao hơn thì sẽ cần nhiều ca kiểm thử hơn các module/hàm có chỉ số McCabe nhỏ hơn
 - Rõ ràng là mỗi một hàm cần có số lượng test case tối thiểu bằng với chỉ số McCabe của hàm đó.
 - Nói một cách khác, với mỗi một hàm chúng ta phải xây dựng các trường hợp kiểm thử sao cho có thể kiểm thử tất cả các nhánh của hàm

31

McCabe Cyclomatic Number - Example

- Đếm số lệnh IFs and LOOPS
- IF: 2, LOOP: 2
- $v(G) = 5$

```
void sort ( int *a, int n ) {
    int i, j, t;

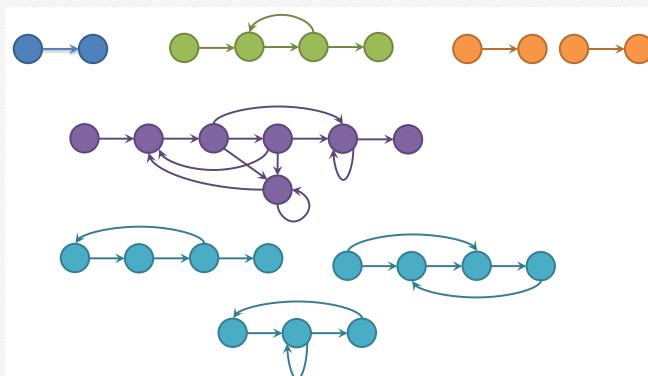
    if ( n < 2 ) return;
    for ( i=0 ; i < n-1; i++ ) {
        for ( j=i+1 ; j < n ; j++ ) {
            if ( a[i] > a[j] ) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}
```

$V = 80 \log_2(24) \approx 392$

32

Other examples based on CFG

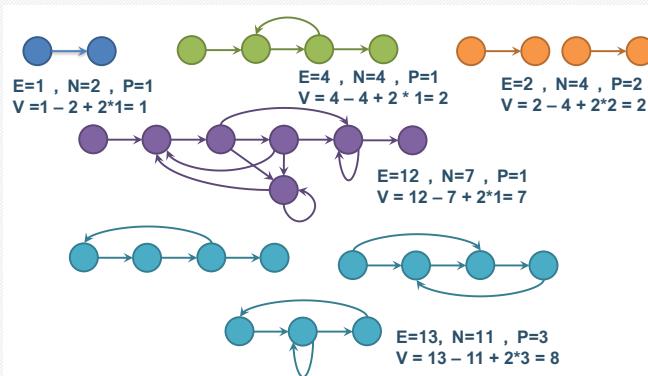
33



33

Other examples based on CFG

34



34

Other examples based on source code

35

```

1 procedure SpeakingTimeToday (in listOfCalls; out speakingTime);
2 begin
3     callNum = 0
4     speakingTime = 0
5     while (callNum < listOfCalls.Length)
6         if listOfCalls[callNum].date = today
7             speakingTime = speakingTime + listOfCalls[CallNum].time
8         end
9         callNum = callNum + 1
10    end
11 end

```

35

Other ex

7

```

1 public class MyBubbleSort {
2
3     public static void bubble_srt(int array[]) {
4         int n = array.length;
5         int k;
6         for (int m = n; m >= 0; m--) {
7             for (int i = 0; i < m - 1; i++) {
8                 k = i + 1;
9                 if (array[i] > array[k]) {
10                     swapNumbers(i, k, array);
11                 }
12             }
13         }
14     }
15
16     private static void swapNumbers(int i, int j, int[] array) {
17         int temp;
18         temp = array[i];
19         array[i] = array[j];
20         array[j] = temp;
21     }
22 }

```

37

McCabe Cyclomatic Number – Recommendations Khuyến cáo đối với chỉ số McCabe

39

- Chỉ số McCabe của một hàm số nên nhỏ hơn 15.
 - Khi một hàm có chỉ số McCabe = 15 ít nhất có 15 nhánh rẽ trong chương trình
- Những chương trình nhiều hơn 15 nhánh rẽ sẽ khó để hiểu và khó để kiểm thử
- Chỉ số McCabe của một file mã nguồn nên nhỏ hơn 100

39

Maintainability Index (MI) Chỉ số bảo trì được của chương trình

40

- Chỉ số bảo trì được của chương trình được tính toán dựa trên các độ đo về số dòng mã nguồn, chỉ số McCabe và các độ đo của Halstead.
- Chỉ số này chỉ ra liệu rằng việc viết lại mã nguồn chương trình có đỡ tốn kém hơn hoặc rủi ro hơn việc thay đổi nó không?
- Có hai phiên bản khác nhau của MI:
 - MI có tính toán số dòng comment
 - MI không tính số dòng comment
- Thực tế có 3 loại số đo khác nhau đối với chỉ số này
 - MIwoc: chỉ số không có comment
 - Mlcw: chỉ số comment
 - MI = MIwoc + Mlcw

40

MI's formula Công thức tính chỉ số MI

41

- $MI_{woc} = 171 - 5.2 * \ln(aveV) - 0.23 * aveG - 16.2 * \ln(aveLOC)$
 - aveV = average Halstead Volume V per module
 - aveG = average extended cyclomatic complexity v(G) per module
 - aveLOC = average count of lines LOCphy per module
 - perCM = average percent of lines of comments per module
- $MI_{cw} = 50 * \sin(\sqrt{2},4 * perCM)$

41

MI Recommendations Các khuyến cáo đối với chỉ số MI

42

- Chỉ số MI của một chương trình với comment nếu:
 - lớn hơn 85: khả năng thay đổi và bảo trì tốt
 - từ 65 đến 85: khả năng bảo trì trung bình
 - <65: khó để bảo trì và thực sự chứa các đoạn mã nguồn không tốt
- Các đoạn mã nguồn lớn, không được chú thích hoặc không được cấu trúc hoá có thể dẫn tới các giá trị MI nhỏ hơn 0 .
- Những đoạn chương trình đó rất cần được cải thiện để nâng cao giá trị MI.

42

Maintainability Index Example

- Halstead's $V = 392$
- McCabe's $v(G) = 5$
- LOC = 14
- MI = 96 --> easy to maintain!

```
void sort ( int *a, int n ) {
    int i, j, t;

    if ( n < 2 ) return;
    for ( i=0 ; i < n-1; i++ ) {
        for ( j=i+1 ; j < n ; j++ ) {
            if ( a[i] > a[j] ) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
        }
    }
}
```

$V = 80 \log_2(24) \approx 392$

Exercise: Calculate the MI

44

1. Draw the control flow graph
2. Calculate McCabe Cyclomatic
3. Calculate MI

```
1 public class BinarySearch {
2     public int binsearch(int x, int[] V, int n) {
3         int low, high, mid;
4         low = 0;
5         high = n - 1;
6         while (low <= high) {
7             mid = (low + high) / 2;
8             if (x < V[mid])
9                 high = mid - 1;
10            else if (x > V[mid])
11                low = mid + 1;
12            else
13                return mid;
14        }
15        return -1;
16    }
17 }
```