1-4 基本的算法结构

魏恒峰

hfwei@nju.edu.cn

2017 年 11 月 06 日

Longest Monotone Subsequence

ES 24.8: Longest Monotone Subsequence

Write a computer program that takes as its input a sequence of distinct integers and returns as its output the length of a longest monotone subsequence.

ES 24.8: Longest Monotone Subsequence

Write a computer program that takes as its input a sequence of distinct integers and returns as its output the length of a longest monotone subsequence.

Understanding this problem:

ES 24.8: Longest Monotone Subsequence

Write a computer program that takes as its input a sequence of distinct integers and returns as its output the length of a longest monotone subsequence.

Understanding this problem:

Subsequence *vs.* substring

ES 24.8: Longest Monotone Subsequence

Write a computer program that takes as its input a sequence of distinct integers and returns as its output the length of a longest monotone subsequence.

Understanding this problem:

Subsequence *vs.* substring

Monotone increasing *vs.* decreasing

ES 24.8: Longest Monotone Subsequence

Write a computer program that takes as its input a sequence of distinct integers and returns as its output the length of a longest monotone subsequence.

Understanding this problem:

Subsequence *vs.* substring

Monotone increasing *vs.* decreasing strictly *vs.* non-strictly

ES 24.8: Longest Monotone Subsequence

Write a computer program that takes as its input a sequence of distinct integers and returns as its output the length of a longest monotone subsequence.

Understanding this problem:

Subsequence *vs.* substring

Monotone increasing *vs.* decreasing   strictly *vs.* non-strictly

Longest existence? uniqueness?

ES 24.8: Longest Monotone Subsequence

Write a computer program that takes as its input a sequence of distinct integers and returns as its output the length of a longest monotone subsequence.

Understanding this problem:

Subsequence *vs.* substring

Monotone increasing *vs.* decreasing   strictly *vs.* non-strictly

Longest existence? uniqueness?

The Length *vs.* the subsequence itself

ES 24.8: Longest (Strictly) Increasing Subsequence (LIS)

- Given an integer array $A[0 \ldots n-1]$
- To find the length L of an LIS

$$0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15 \implies 0, 2, 6, 9, 11, 15$$

ES 24.8: Longest (Strictly) Increasing Subsequence (LIS)

- Given an integer array $A[0 \ldots n-1]$
- To find the length L of an LIS

学生反馈：这道题为什么放在 "Pigeonhole Principle" 这一章？

学生反馈： 这道题为什么放在 “Pigeonhole Principle” 这一章？

学生反馈： 这道题为什么放在 “Pigeonhole Principle” 这一章?

Theorem (Erdős-Szekeres Theorem)

Let n be a positive integer. Every sequence of n^2+1 distinct integers must contain a monotone subsequence of length $n+1$.

Q：这道题与（强）数学归纳法有什么关系？

Q : 这道题与 (强) 数学归纳法有什么关系?

B.S. $P(0)$

I.H. $P(0) \cdots P(i-1)$

I.S. $P(0) \cdots P(i-1) \to P(i)$

Q : 这道题与 (强) 数学归纳法有什么关系?

B.S. $P(0)$

I.H. $P(0) \cdots P(i-1)$

I.S. $P(0) \cdots P(i-1) \to P(i)$

$P(i)$ 是什么?

$\boxed{P(i) : \text{the length of an LIS in } A[0 \cdots i].}$

$\boxed{P(i) : \text{the length of an LIS in } A[0 \cdots i].}$

$$L = P(n-1)$$

$\boxed{P(i) : \text{the length of an LIS in } A[0 \cdots i].}$

$$L = P(n-1)$$

$$P(0) = 1$$

$\boxed{P(i) : \text{the length of an LIS in } A[0 \cdots i].}$

$$L = P(n-1)$$

$$P(0) = 1$$

$$P(0) \cdots P(i-1) \to P(i)?$$

$\boxed{P(i) : \text{the length of an LIS in } A[0 \cdots i].}$

$$L = P(n-1)$$

$$P(0) = 1$$

$$P(0) \cdots P(i-1) \to P(i)?$$

$$P(i) = \max\{P(i-1), \max_{\substack{0 \le j < i \\ A[j] < A[i]}} \{P(j) + 1\}\}$$

$\boxed{P(i) : \text{the length of an LIS in } A[0 \cdots i].}$

$$L = P(n-1)$$

$$P(0) = 1$$

$$P(0) \cdots P(i-1) \to P(i)?$$

$$P(i) = \max\{P(i-1), \max_{\substack{0 \le j < i \\ A[j] < A[i]}} \{P(j) + 1\}\}$$

$\boxed{P(i) : \text{the length of an LIS in } A[0 \cdots i].}$

$$L = P(n-1)$$

$$P(0) = 1$$

$$P(0) \cdots P(i-1) \to P(i)?$$

$$P(i) = \max\{P(i-1), \max_{\substack{0 \le j < i \\ A[j] < A[i]}} \{P(j) + 1\}\}$$

$\boxed{P(i) : \text{the length of an LIS } \textit{ending at } A[i].}$

$\boxed{P(i) : \text{the length of an LIS } \textit{ending at } A[i].}$

$$L = \max_{0 \le i < n} P(i)$$

$\boxed{P(i) : \text{the length of an LIS } \textit{ending at } A[i].}$

$$L = \max_{0 \le i < n} P(i)$$

$$P(0) = 1$$

$\boxed{P(i) : \text{the length of an LIS } \textit{ending at } A[i].}$

$$L = \max_{0 \le i < n} P(i)$$

$$P(0) = 1$$

$$P(0) \cdots P(i-1) \to P(i)?$$

$\boxed{P(i) : \text{the length of an LIS } \textit{ending at } A[i].}$

$$L = \max_{0 \le i < n} P(i)$$

$$P(0) = 1$$

$$P(0) \cdots P(i-1) \to P(i)?$$

$$P(i) = \max_{\substack{0 \le j < i \\ A[j] < A[i]}} \{P(j) + 1\}$$

```
P(0) = 1;

for (int i = 1; i < n; ++i)
  P(i) = max_{0≤j<i, A[j]<A[i]} {P(j) + 1}

  return L = max_{0≤i<n} P(i);
```

$$P(0) = 1;$$

$$P(i) = \max_{\substack{0 \le j < i \\ A[j] < A[i]}} \{P(j) + 1\}$$

$$L = \max_{0 \le i < n} P(i);$$

```
P(0) = 1;

for (int i = 1; i < n; ++i)   // How much time?
    P(i) = max_{0≤j<i, A[j]<A[i]} {P(j) + 1}

    return L = max_{0≤i<n} P(i);
```

$$P(0) = 1;$$

$$P(i) = \max_{\substack{0 \le j < i \\ A[j] < A[i]}} \{P(j) + 1\}$$

$$L = \max_{0 \le i < n} P(i);$$

$P(0) = 1$;

for (**int** i = 1; i < n; ++i) // How much time?

$$P(i) = \max_{\substack{0 \le j < i \\ A[j] < A[i]}} \{P(j) + 1\}$$

return $L = \max_{0 \le i < n} P(i)$; // How much space?

$P(0) = 1;$

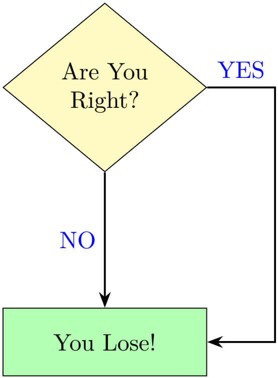
```
for (int i = 1; i < n; ++i)   // How much time?
```

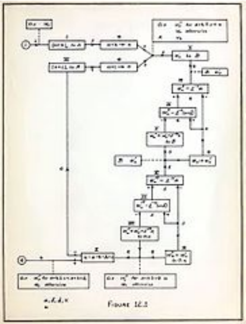
$$P(i) = \max_{\substack{0 \le j < i \\ A[j] < A[i]}} \{P(j) + 1\}$$

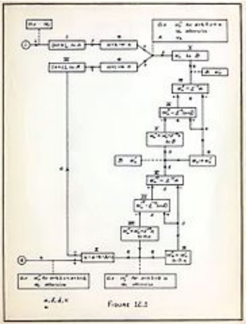
return $L = \max_{0 \le i < n} P(i);$ `// How much space?`

Flowcharts

How to Argue with Your Girlfriend?





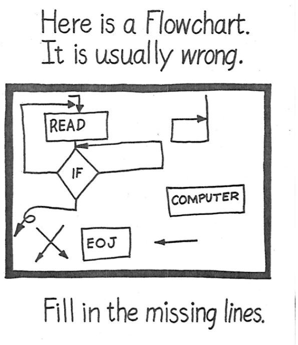


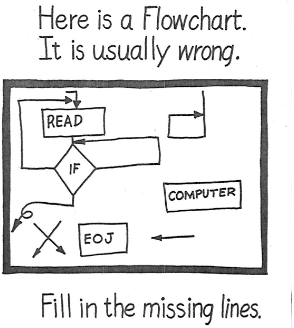
We feel certain that a moderate amount of experience with this stage of coding suffices to remove from it all difficulties, and to make it a perfectly routine operation.

— John von Neumann and Herman Goldstine, late 1940s


我的内心几
乎是崩溃的

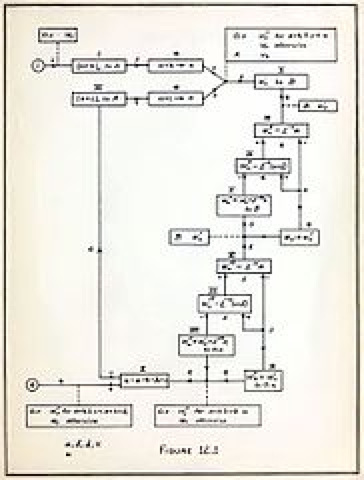

我的内心几
乎是崩溃的


Here is a Flowchart.
It is usually wrong.
READ
IF
COMPUTER
EOJ
Fill in the missing lines.



Any resemblance between our flow charts and the present program is purely coincidental.

— *Donald Knuth, 1963*



Flowcharts Considered Harmful.

Just my opinion...

Just my opinion...

Draw it when it does help

Just my opinion...

Draw it when it does help

OR you have to.

Simulations

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(a) "`for-do`" by "`while-do`"

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(a) "for-do" by "while-do"

```
for (int i = 0; i < N; ++i)
  statement
```

```
int i = 0;
while (i < N)
  statement
  ++i
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(a) "`for-do`" by "`while-do`"

```
for (int i = 0; i < N; ++i) // not general!
  statement
```

```
int i = 0;
while (i < N)
  statement
  ++i
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(a) "`for-do`" by "`while-do`"

```
for (init; cond; inc)
  statement
```

```
init;
while (cond)
  statement
  inc
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(a) "`for-do`" by "`while-do`"

```
for (init; cond; inc)
  statement
```

```
init;
while (cond)
  statement
  inc
```

Whether to use "while" or "for" is largely a matter of personal preference.

— K&R C Bible

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(b) "`if-then` & `if-then-else`" by "`while-do`"

```
if (A)
  B
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(b) "`if-then` & `if-then-else`" by "`while-do`"

```
while (A)
  B
  ¬ A
```

```
if (A)
  B
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(b) "`if-then` & `if-then-else`" by "`while-do`"

```
if (A)
  B
```

```
while (A)
  B
  ¬ A // Wrong: side effects?
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(b) "`if-then` & `if-then-else`" by "`while-do`"

```
if (A)
  B
```

```
while (A)
  B
  ¬ A // Wrong: side effects?
```

```
flag = 1
while (A && flag)
  B
  flag = 0
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(b) "`if-then` & `if-then-else`" by "`while-do`"

```
if (A)
  B
else
  C
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(b) "`if-then` & `if-then-else`" by "`while-do`"

```
if (A)
  B
else
  C
```

```
flag_if = 1
while (A && flag_if)
  B
  flag_if = 0
flag_else = 1
while (¬ A && flag_else)
  C
  flag_else = 0
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(b) "`if-then` & `if-then-else`" by "`while-do`"

```
if (A)
  B
else
  C
```

```
flag_if = 1
while (A && flag_if)
  B  // Wrong: side effects?
  flag_if = 0
flag_else = 1
while (¬ A && flag_else)
  C
  flag_else = 0
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(b) "`if-then` & `if-then-else`" by "`while-do`"

```
if (A)
  B
else
  C
```

```
flag_if = 1
while (A && flag_if)
  B  // Wrong: side effects?
  flag_if = 0
flag_else = 1
while (¬ A && flag_else)
  C
  flag_else = 0
```

```
flag = 1
while (A && flag)
  B
  flag = 0
while (¬ A && flag)
  C
  flag = 0
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(c) "`while-do`" by "`if-then & goto`"

(d) "`while-do`" by "`repeat-until & if-then`"

```
while (A)
  B
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(c) "`while-do`" by "`if-then & goto`"

(d) "`while-do`" by "`repeat-until & if-then`"

```
while (A)
  B
```

```
L: if (A)
     B
     goto L
```

DH 2.5: Simulations

Perform the following simulations of some control constructs by others.

(c) "while-do" by "if-then & goto"

(d) "while-do" by "repeat-until & if-then"

```
while (A)
    B
```

```
L: if (A)
      B
      goto L
```

```
if (A)
  repeat
    B
  until (¬ A)
```

DH 2.8: Simulations

Simulate "`while-do`" by "`if-then-else & recursive`".

```
while (A)
  B
```

DH 2.8: Simulations

Simulate "`while-do`" by "`if-then-else & recursive`".

```
while (A)
  B
```

```
simulateWhile() {
  if (A)
    B
    simulateWhile();

  return;
}
```

DH 2.8: Simulations

Simulate “`while-do`” by “`if-then-else & recursive`”.

```
while (A)
  B
```

```
simulateWhile() { // define function
  if (A)
    B
    simulateWhile();

  return;
}
```

DH 2.8: Simulations

Simulate "`while-do`" by "`if-then-else & recursive`".

```
while (A)
  B
```

```
simulateWhile() { // define function
  if (A)
    B
    simulateWhile();

  return;
}
```

(1) `A;B`

(2) `if-then`

(3) `if-then-else`

(4) `for-do`

(5) `while-do`

(6) `repeat-until`

(1) A;B

(2) if-then

(3) if-then-else

(4) for-do

(5) while-do

(6) repeat-until

```
repeat
  B
until (¬ A)
```

(1) A;B

(2) if-then

(3) if-then-else

(4) for-do

(5) while-do

(6) repeat-until

```
repeat
  B
until (¬ A)
```

```
B
while (A)
  B
```

(1) `A;B`

(2) `if-then`

(3) `if-then-else`

(4) `for-do`

(5) `while-do`

(6) `repeat-until`

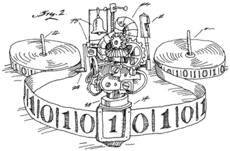
```
repeat
  B
until (¬ A)
```

```
B
while (A)
  B
```

Theorem ("On Folk Theorems" (David Harel, 1980))

Any computable function can be computed by a "while-do" (and ";") program (with additional Boolean variables).

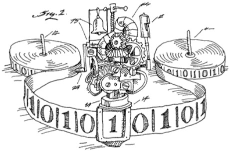

i ♥ minimalism

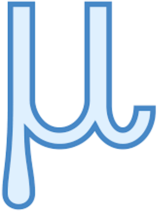


λ

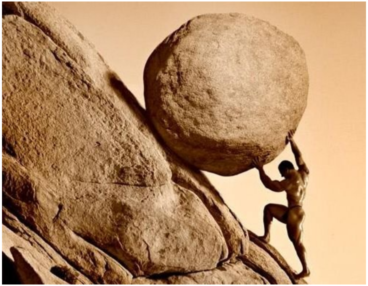
μ

Simulations for Equivalence

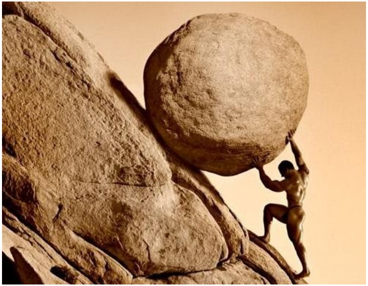




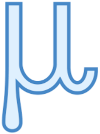
Bounded Iterations *vs.* Unbounded Iterations



Bounded Iterations *vs.* Unbounded Iterations

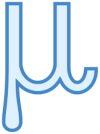


Q : Why unbounded iterations?



μ-Recursive Functions

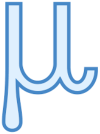
$$\color{red}{\mu}\color{black} y(g(x,y)) = \Big(\operatorname*{argmin}_{y} g(x,y) = 0 \Big)$$



μ-Recursive Functions

$$\color{red}{\mu} y(g(x,y)) = \left(\operatorname*{argmin}_y g(x,y) = 0\right)$$

Unbounded iterations: “`while-do`”



μ-Recursive Functions

$$\color{red}{\mu} y(g(x,y)) = \left(\operatorname*{argmin}_y g(x,y) = 0\right)$$

Unbounded iterations: "`while-do`"

Theorem (Ackermann Function)

The Ackermann function is μ-recursive but not primitive recursive (which contains bounded iterations.).

DH 2.4: Bounded Iteration

Given a list L of N integers, to produce in S and P the sum of the even numbers in L and the product of the odd ones, respectively.

DH 2.4: Bounded Iteration

Given a list L of N integers, to produce in S and P the sum of the even numbers in L and the product of the odd ones, respectively.

```
int S = 0, P = 1;
for (int i = 0; i < N; ++i) {
  if (L(i) % 2 == 0)
    S += L(i);
  else
    P *= L(i);
}
```

DH 2.4: Bounded Iteration

Given a list L of N integers, to produce in S and P the sum of the even numbers in L and the product of the odd ones, respectively.

```
int S = 0, P = 1;
for (int i = 0; i < N; ++i) {
  if (L(i) % 2 == 0)
    S += L(i);
  else
    P *= L(i);
}
```

DH 2.1: Salary Summation

$N - 1$ vs. N iterations

DH 2.4: Bounded Iteration

Given a list L of N integers, to produce in S and P the sum of the even numbers in L and the product of the odd ones, respectively.

```
int S = 0, P = 1;
for (int i = 0; i < N; ++i) {
  if (L(i) % 2 == 0)
    S += L(i);
  else
    P *= L(i);
}
```

DH 2.1: Salary Summation

$N - 1$ vs. N iterations

DH 2.7: Compute $n!$

Write algorithms that compute $n!$, given a non-negative integer n.

(a) Using iteration statements.

(b) Using recursion.

DH 2.7: Compute $n!$

Write algorithms that compute $n!$, given a non-negative integer n.

(a) Using iteration statements.

(b) Using recursion.

```
int P = 1;
for (int i = 2; i <= n; ++i) {
  P *= i;
}
```

DH 2.7: Compute $n!$

Write algorithms that compute $n!$, given a non-negative integer n.

(a) Using iteration statements.

(b) Using recursion.

```
int P = 1;
for (int i = 2; i <= n; ++i) {
  P *= i;
}
```

```
int recursive-factorial(int n) {
  if (n == 1)
    return 1;

    else return n * recursive-factorial(n-1);
}
```

DH 2.7: Compute $n!$

Write algorithms that compute $n!$, given a non-negative integer n.

(a) Using iteration statements.

(b) Using recursion.

```
int P = 1;
for (int i = 2; i <= n; ++i) {
  P *= i;
}
```

```
int recursive-factorial(int n) { // define function
  if (n == 1)
    return 1;

    else return n * recursive-factorial(n-1);
}
```

DH 2.7: Compute $n!$

Write algorithms that compute $n!$, given a non-negative integer n.

(a) Using iteration statements.

(b) Using recursion.

```
int P = 1;
for (int i = 2; i <= n; ++i) {
  P *= i;
}
```

```
int recursive-factorial(int n) { // define function
  if (n == 1)
    return 1;
    // NOT: return n * (n - 1)!
    else return n * recursive-factorial(n-1);
}
```

Stack and Permutations


Thank You!