Pseudo-code Analysis

Richard Hua

Semester 2, 2021

Running time of pseudo-code

- Need to be careful what we are counting.
- Specify, specify, specify...
- Can get complicated.

Running time of pseudo-code

General strategy:

- Running time of disjoint blocks adds.
- Running time of nested loops with non-interacting variables multiplies.
- If the structure is more complicated, find out how many iterations each loop has first (if possible), and then analyze how the loops/condition checks interact with each other to decide.

Simple pseudo-code

Required: $0 \leqslant i \leqslant j \leqslant n-1$

function SWAP(array $a[0 \cdots n-1]$, integer i, integer j)

$t \leftarrow a[i]$

$a[i] \leftarrow a[j]$

$a[j] \leftarrow t$

return a

If only count value assignment as elementary operations, then 3 operations.

Algorithm with loops

function SUM(array $a[0 \cdots n-1]$)

  $k \leftarrow 0$

  for $j \leftarrow 0$ to $n-1$ **do**

    $k+ = a[j]$

  return k

Be careful ....

Algorithm with loops

Same algorithm in Python-style pseudo-code

function SUM(array $a[0 \cdots n-1]$)
  $k \leftarrow 0$
  for j in range(n) **do**
    $k+ = a[j]$
  return k

Algorithm with loops

Same algorithm in Java-style pseudo-code

function SUM(array $a[0 \cdots n-1]$)
    $k \leftarrow 0$
    for (int $j = 0$; $j < n$; $j++$)
        $k += a[j]$
    return k

Question

Do we count the operations we need for the loop-counter variable?

Algorithm with loops

Same algorithm in Python-style pseudo-code

function SUM(array $a[0 \cdots n-1]$)
  $k \leftarrow 0$
  for j in range(n) **do**
    $k += a[j]$
  return k

Question: Do we count those operations in this case? Does it even use these variable?
Answer: I don't know, it probably does in the background (for an actual Python program).
Conclusion: Specify, specify, specify....
Let's say we only count arithmetic operations inside the main loop.

Algorithm with loops

function SUM(array $a[0 \cdots n-1]$)
  $k \leftarrow 0$
  for j in range(n) **do**
    $k+ = a[j]$
  return k

j-loop has n iterations, 1 addition per iteration.

Trickier example

function MEH(integer n)
$\quad i \leftarrow 1$
$\quad$ **while** $i < n$ **do**
$\quad\quad$ print i
$\quad\quad i \leftarrow 2i$

Count all operations including value assignments, arithmetic operations and condition checks.

Nested loops

function MEH(array $a[0 \cdots n-1]$, $b[0 \cdots n-1]$)
  $t \leftarrow 0$
  for i in range(n) **do**
    for j in range(n) **do**
      $t = t + a[i] * b[j]$
  return t

Suppose we are only counting arithmetic operations inside the j loop.
The loops have independent variables.
They both have n iterations.
Total number of iteration is n^2, 2 operations per iteration.

Nested loops

function MEH(integer n)
  for i in range(n) **do**
    for j in range(i, n) **do**
      print $i + j$

Number of iteration of the j loop depends on the value of i.
Can't just simply multiply them.

Nested loops

function MEH(integer n)
  for i in range(n) **do**
    for j in range(i, n) **do**
      print $i + j$

Value of i goes from $0, 1, \cdots n - 1$.
For each value of i, the j loop has $n - i$ iterations.
So the total number is $\sum_{i=1}^{i \leqslant n} i = \frac{n(n+1)}{2}$.

Hard examples

for $i = 1$; $i < n$; $i \leftarrow 2i$ **do**
    for $j = 1$; $j < n$; $j \leftarrow 2j$ **do**
        if $j = 2i$ **then**
            for $k = 0$; $k < n$; $k \leftarrow k + 1$ **do**
                constant number of operations
        else
            for $k = 1$; $k < n$; $k \leftarrow 3k$ **do**
                constant number of operations

Suppose we only count number of operations inside the two k-loops.

Hard examples

for $i = 1;\ i < n;\ i \leftarrow 2i$ **do**
  for $j = 1;\ j < n;\ j \leftarrow 2j$ **do**
    if $j = 2i$ **then**
      for $k = 0;\ k < n;\ k \leftarrow k + 1$ **do**
        constant number of operations
    else
      for $k = 1;\ k < n;\ k \leftarrow 3k$ **do**
        constant number of operations

i-loop and j-loop both have $\lceil \lg n \rceil$ iterations. The first k-loop has n iterations, the second has $\lceil \log_3 n \rceil$ iterations. The *if* condition can be true for only 1 iteration of the j-loop (during execution of the j-loop, the value of i is fixed) except for the last iteration of the i-loop (say $i = 2^{k-1}$ when $n = 2^k$) when no value of j is $2i$.

Hard examples

for $i = 1;\ i < n;\ i \leftarrow 2i$ **do**
  for $j = 1;\ j < n;\ j \leftarrow 2j$ **do**
    if $j = 2i$ **then**
      for $k = 0;\ k < n;\ k \leftarrow k + 1$ **do**
        constant number of operations
    else
      for $k = 1;\ k < n;\ k \leftarrow 3k$ **do**
        constant number of operations

So a complete expression for everything inside the j-loop (except for the last iteration of the i-loop) is $cn + c(\lceil \lg n \rceil - 1) \log_3 n$, assuming we do c operations inside both k-loops. This is repeated $\lceil \lg n \rceil - 1$ times for the first $\lceil \lg n \rceil - 1$ iterations of the $i - loop$ so we have $c(\lceil \lg n \rceil - 1)(n + (\lceil \lg n \rceil - 1) \log_3 n)$. For the last iteration of the i-loop, we did $c \lceil \lg n \rceil \lceil \log_3 n \rceil$ operations so the complete expression is $c(\lceil \lg n \rceil - 1)(n + (\lceil \lg n \rceil - 1) \log_3 n + c \lceil \lg n \rceil \lceil \log_3 n \rceil$.

Hard examples

$m \leftarrow 2$
for $j = 1$; $j <= n$; $j \leftarrow j + 1$ **do**
    if $j = m$ **then**
        $m \leftarrow 2m$
        for $i = 1$; $i <= n$; $i \leftarrow i + 1$ **do**
            constant number operations

We leave this one as an exercise.

Example

What is the running time of the following piece of pseudo-code?

function SEARCH(array $a[0 \cdots n-1]$, integer k)
  for $i = 0$; $i < n$; $i \leftarrow i + 1$ **do**
    if $a[i] = k$
      return i
  return -1

How many iterations will the for loop run?
Impossible to tell without knowing at least some information about the input (e.g. distribution of variables etc.).
Seems input dependent.

Analysis by case

- Best case. Minimum number of operations.
- Worst case. Maximum number of operations.
- Average case. Average of all cases.

Best case

- Quite boring.
- Often quite good.
- Mostly useless since it includes trivial cases where there is no difficulties at all.

Worst case

- Very important in some cases. Worst-case bounds are valid for all inputs so it's especially important for mission-critical component of an application.
- Often not too hard to derive mathematically.
- It might be just some very specific inputs that don't really appear in practice. Often hugely exceed typical running time and therefore have little predictive or comparative value.
- Even if it's bad, it doesn't mean your algorithm is useless.

Average case

- Average of all cases, gives us some understandings as to how the algorithm will perform in practice when input is random.
- Hard to perform.
- Do we take all possible inputs? Or restrict us to exclude 'artificial' inputs. If so, how do we define 'artificial'?
- Do all inputs appear with the same probability? If not, what distribution do they follow?
- Often involves difficult mathematical evaluations.

Analysis by case

Conclusion: a good worst-case bound is always useful, but it is just a first step and we should aim to refine the analysis for important algorithms. Average-case analysis is often more practically useful, provided the algorithm will run on 'random' data and we have some tolerance for risk.

Note that the different cases are algorithm-specific, not problem specific.

Best case

```
function SEARCH(array a[0 ··· n − 1], integer k)
    for i = 0; i < n; i ← i + 1 do
        if a[i] = k
            return i
    return −1
```

k is the first element of a, 1 comparison.

Worst case

function SEARCH(array $a[0 \cdots n-1]$, integer k)
  for $i = 0$; $i < n$; $i \leftarrow i + 1$ **do**
    if $a[i] = k$
      return i
  return -1

k is not in a, need to loop through all elements of a so n comparisons.

Average case

function SEARCH(array $a[0 \cdots n-1]$, integer k)
  for $i = 0$; $i < n$; $i \leftarrow i + 1$ **do**
    if $a[i] = k$
      return i
  return -1

Assume that the first instance of the integer k we are looking for has the same probability of appearing at each of the n indexes of the array or not in a at all.

There are $n+1$ possible choices for the first occurrence of k in a (e.g. at index $0, 1, 2, \ldots n-1$ or not in a at all). Each of them require $1, 2, 3, \cdots, n, n$ comparisons respectively.

Number of comparisons overall cases is $\frac{n(n+1)}{2} + n = \frac{n(n+3)}{2}$. So the average is $\frac{n(n+3)}{2(n+1)}$.

As n gets large, $\frac{n(n+3)}{2(n+1)}$ is approximately $n/2$.

Things can still get more complicated

- It's a bit tedious to workout the exact number of operations.
- Doesn't affect the result much in most cases.
- Recursive algorithms.

Conclusion: Need more and better tools!

Recursive algorithms

- Very commonly seen in practice.
- Easy to implement (depending on how the problem is defined).
- Not a good idea to use in a lot of cases. E.g. SLOWFIB.

Recursive algorithms

Recall the SLOWFIB pseudo-code.

function SLOWFIB(integer n)
    if $n < 0$ then return 0
    else if $n = 0$ or $n = 1$ then return n
    else return SLOWFIB(n-1) + SLOWFIB(n-2)

Suppose we want to calculate the number of additions needed.

Recursive algorithms

Recall the SLOWFIB pseudo-code.

function SLOWFIB(integer n)
    if $n < 0$ then return 0
    else if $n = 0$ or $n = 1$ then return n
    else return SLOWFIB(n-1) + SLOWFIB(n-2)

Define function $T(n)$ as the number of additions needed by SLOWFIB to calculate $F(n)$.
When $n = 0$ or 1, there is no addition. Base case of our recursive algorithm. $T(0) = T(1) = 0$.
Otherwise, we need to recursively calculate how many additions are needed to compute $T(n-1)$ and $T(n-2)$ and then 1 more addition is needed to add them together. So
$T(n) = T(n-1) + T(n-2) + 1$ if $n > 1$.