Running time analysis

Lecture 10

Measuring the running “time”

• Goal: Analyze an algorithm written in pseudocode and describe its running time
  – Without having to write code
  – In a way that is independent of the computer used

• To achieve that, we need to
  – Make simplifying assumptions about the running time of each basic (primitive) operations
  – Study how the number of primitive operations depends on the size of the problem solved

Primitive operation

Simple computer operation that can be performed in time that is always the same, independent of the size of the bigger problem solved (we say: constant time)

- Assigning a value to a variable: \( x \leftarrow 1 \)
- Calling a method: Expos.addWin()
  – Note: doesn’t include the time to execute the method
- Returning from a method: return \( x \)
- Arithmetic operations on primitive types
  \( x + y, x * 3.1416, x / y \), etc.
- Comparisons on primitive types: \( x \sim y \)
- Conditionals: if (...) then.. else...
- Indexing into an array: \( A[i] \)
- Following object reference: Expos.losses

• Note: Multiplying two bigIntegers is not a primitive operation, because the running time depends on the size of the numbers multiplied.

FindMin analysis

Algorithm findMin(A, start, stop)
Input: An array A and indices start and stop
Output: The index of the smallest element of A between start and stop (inclusively)

\[
\begin{align*}
\text{minvalue} & \leftarrow A[\text{start}] \\
\text{minindex} & \leftarrow \text{start} \\
\text{index} & \leftarrow \text{start} + 1 \\
\text{while } (\text{index } \leq \text{stop}) \text{ do } \\
& \quad \text{if } (A[\text{index}] < \text{minvalue}) \\
& \quad \quad \text{minvalue } \leftarrow A[\text{index}] \\
& \quad \quad \text{minindex } \leftarrow \text{index} \\
& \quad \text{index } \leftarrow \text{index } + 1 \\
\text{return } \text{minindex}
\end{align*}
\]

Running time

\[
T_{\text{index}} + T_{\text{assign}} + T_{\text{call}} + T_{\text{assign}} + T_{\text{arith}} + T_{\text{assign}} + T_{\text{arith}} + T_{\text{call}} + T_{\text{assign}} + T_{\text{arith}} + T_{\text{assign}} + T_{\text{arith}}
\]

Best case, Worst case

• Running time depends on \( n = \text{stop} - \text{start} \)
  – But it also depends on the content of the array!
• What kind of array of \( n \) elements will give the best running time for findMin?
  – The worst running time?

More assumptions

• Counting each type of primitive operations is tedious
• We assume that the running time of each operation is roughly comparable:
  \( T_{\text{assign}} \approx T_{\text{comp}} \approx T_{\text{arith}} \approx \ldots \approx T_{\text{index}} = 1 \) primitive operation
• We are only interested in the number of primitive operations performed

Worst-case running time for findMin becomes:
Selection Sort

```
Algorithm SelectionSort(A,n)
Input: an array A of n elements
Output: the array is sorted
i← 0
while (i<n) do {
    minindex ← findMin(A,i,n-1)
    t ← A[minindex]
    A[i] ← t
    i← i + 1
}
```

Primitive operations (worst case):
1. \(\sum_{i=0}^{n-1} i\) times
2. \(\sum_{i=0}^{n-1} i\) times

Total: \(T(n) = 1 + \left(\sum_{i=0}^{n-1} 20 + 10 (n - 1 - i)\right)\)

Simplification #1:
When \(n\) is large, \(T(n) \approx 5n^2\)

Simplification #2:
When \(n\) is large, \(T(n)\) grows approximately like \(n^2\)
We will write \(T(n)\) is \(O(n^2)\)
“\(T(n)\) is big-O of \(n\) squared”

More simplifications

We have: \(T(n) = 1 + 15n + 5n^2\)

Simplification #1:
When \(n\) is large, \(T(n) \approx 5n^2\)

Simplification #2:
When \(n\) is large, \(T(n)\) grows approximately like \(n^2\)
We will write \(T(n)\) is \(O(n^2)\)
“\(T(n)\) is big-O of \(n\) squared”