COMP250: Running time analysis

Jérôme Waldispühl
School of Computer Science
McGill University

Based on slides from M. Blanchette

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 Operations

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 ←1 — T_{assign}
- **Calling a method:** Expos.addWin() — T_{call}
  - Note: doesn't include the time to execute the method
- **Returning from a method**: return x; — T_{return}
- **Arithmetic operations on primitive types** — T_{arith}
  x + y, r*3.1416, x/y, etc.
- **Comparisons on primitive types:** x==y — T_{comp}
- **Conditionals:** if (...) then.. else... — T_{cond}
- **Indexing into an array:** A[i] — T_{index}
- **Following object reference:** Expos.losses — T_{ref}

Note: Multiplying two LargeIntegers 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: Array A, index start & stop
Output: Index of the smallest element of A[start:stop]
minvalue ← A[start]
minindex ← start
index ← start + 1
while ( index <= stop ) do {
   if (A[index]<minvalue)
   then {
        minvalue ← A[index]
        minindex ← index
   }
      index = index + 1
}
return minindex
```

Running time

$T_{index} + T_{assign}$	
T_{assign}	
$T_{arith} + T_{assign}$	
$T_{comp} + T_{cond}$	repeated
$T_{index} + T_{comp} + T_{cond}$	stop-start
$T_{index} + T_{assign}$	times
T_{assign}	
$T_{assign} + T_{arith}$	
$T_{comp} + T_{cond}$ (last check of loop)	
T_{return}	

Worst case running time

- Running time depends on *n* = stop – start + 1
  - But it also depends on the content of the array!
- What kind of array of *n* elements will give the worst running time for findMin?

Example:

5	4	3	2	1	0

- The best running time?

Example:

0	1	2	3	4	5

More assumptions

- Counting each type of primitive operations is tedious
- The running time of each operation is roughly comparable:

$T_{assign} \approx T_{comp} \approx T_{arith} \approx \ldots \approx T_{index}$ = 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[minindex] ← A[i]
    A[i] ← t
  i ← i + 1
}
```

Primitive operations (worst case) :

1

2

$3 + T_{FindMin}(n-1-i+1) = 3 + (10\,(n-i) - 2)$

2

3

2

2

2 (last check of loop condition)

Selection Sort: adding it up

Total: $T(n) = 1 + (\sum_{i=0}^{n-1} 12 + 10 (n - i)) + 2$

$= 3 + (12\ n + 10 \sum_{i=0}^{n-1} (n-i)$

$= 3 + 12\ n + 10 (\sum_{i=0}^{n-1} n) - 10 (\sum_{i=0}^{n-1} i)$

$= 3 + 12\ n + 10\ n * n - 10 ((n-1)*n) / 2)$

$= 3 + 12\ n + 10\ n^2 - 5\ n^2 + 5\ n$

$= 5\ n^2 + 17\ n + 3$

More simplifications

We have: $T(n) = 5\ n^2 + 17\ n + 3$

Simplification #1:

When n is large, $T(n) \approx 5\ n^2$

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"