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 ←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 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)

Code	Running time	
minvalue ← A[start]	$T_{index} + T_{assign}$	
minindex ← start	T_{assign}	
index ← start + 1	$T_{arith} + T_{assign}$	
while (index <= stop) **do {**	$T_{comp} + T_{cond}$	repeated stop-start-1 times
if (A[index] < minvalue) **then {**	$T_{index} + T_{comp} + T_{cond}$	
minvalue ← A[index]	$T_{index} + T_{assign}$	
minindex ← index	T_{assign}	
}		
index = index + 1	$T_{assign} + T_{arith}$	
}		
return minindex	T_{return}	

Best case, Worst case

- Running time depends on *n* = stop-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:

$T_{assign} \approx T_{comp} \approx T_{arith} \approx \ldots \approx T_{index}$ = 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

	Primitive operations (worst case) :	
i← 0	1	
while (i<n) **do {**	2	n times
minindex ← findMin(A,i,n-1)	3 + (6 + 10 (n -1 - i))	n times
t ← A[minindex]	2	n times
A[minindex] ← A[i]	3	n times
A[i] ← t	2	n times
i ← i + 1	2	n times
}		

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

$= 1 + 20 n + 10 (n-1) n - 10 \sum_{i=0}^{n-1} i$

$= 1 + 20 n + 10 (n-1) n - 10 (n-1) n /2$

$= 1 + 20 n + 5 (n-1) n = 1 + 15 n + 5 n^2$

More simplifications

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

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"