CSC165
Mathematical Expression and Reasoning for Computer Science

Module 16

Asymptotic Running Time

Introduction

- Computer scientists talk like…
  - "The worst-case runtime of bubble-sort is in $O(n^2)$."
  - "I can sort it in $n \log n$ time."
  - "That's too slow, make it linear-time."
  - "That problem cannot be solved in polynomial time."

Example

- Compare two sorting algorithms:
  - Bubble sort
  - Merge sort
- Demo at http://www.sorting-algorithms.com
- Observations
  - Merge is faster than bubble
  - With larger input size, the advantage of merge over bubble becomes larger
- When input size grows from 20 to 40:
  - The "running time" of bubble roughly quadrupled
  - The "running time" of merge roughly doubled

Input Size	20	40
Bubble	~8.6 sec	~38.0 sec
Merge	~5.0 sec	~11.2 sec

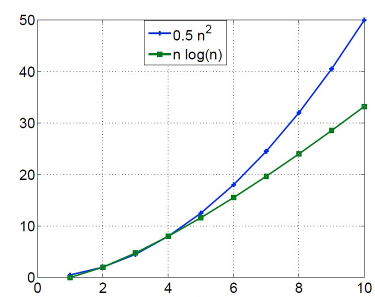
Running Time

- What does "running time" really mean in computer science?
  - It does NOT mean how many seconds are spent in running the algorithm
  - It means the number of steps that are taken by the algorithm
- Running time is independent of the hardware on which you run the algorithm
  - It only depends on the algorithm itself
  - You can run bubble on a super computer and run merge on a mechanical watch. That has nothing to do with the fact that merge is a faster sorting algorithm than bubble

Running Time

- Algorithm running time: number of steps as a function of n, the size of input
- Running time of bubble could be $0.5n^2$ (steps)
- Running time of merge could be $n \log n$ (steps)

Input Size	20	40
Bubble	200 steps	800 steps
Merge	120 steps	295 steps



Asymptotic Running Time

- What we care about is "how the number of steps grows as the size of input grows"
- We care about "when input size doubles, the running time quadruples"
- So, $0.5n^2$ and $700n^2$ are no different!
- Constant factors do NOT matter when it comes to growth!

Example

- Let $T_1(n) = 0.5n^2$
- Let $T_2(n) = 700n^2$
- What happens when n is doubled?
- $\frac{T_1(2n)}{T_1(n)} = \frac{0.5(2n)^2}{0.5n^2} = \frac{0.5(4)n^2}{0.5n^2} = 4$
- $\frac{T_2(2n)}{T_2(n)} = \frac{700(2n)^2}{700n^2} = \frac{700(4)n^2}{700n^2} = 4$

Asymptotic Running Time

- We care about large input sizes
- We do not need to study algorithms in order to sort two elements, because different algorithms make no difference
- We care about algorithm design when the input size n is very large
- So, n^2 and $n^2 + n + 2$ are no different, because when n is really large, $n + 2$ is negligible compared to n^2
- Only the highest-order term matters
- Example:
  - $T_1(n) = n^2$
  - $T_2(n) = n^2 + n + 2$
  - $T_1(10000) = 100{,}000{,}000$
  - $T_2(10000) = 100{,}010{,}002$
  - Difference $\approx 0.01\%$

Summary

- We count the number of steps
- Constant factors do not matter
- Only the highest-order term matters
- Example:
  - The following functions are of the same class
  - n^2 $\quad 2n^2 + 3n$ $\quad \frac{n^2}{165} + 1130n + 3.14159$
  - That class could be called $O(n^2)$
- $O(n^2)$ is an asymptotic notation

Algorithm Complexity

Linear Search

```
def LS(A, x):
""" Return index i, x == A[i].
Otherwise, return -1 """
i = 0
while i < len(A):
    if A[i] == x:
        return i
    i = i + 1
return -1
```

- What is the running time complexity of this program?
  - We count the number of steps
  - Count the number of executed lines of code
  - Cannot say yet… it depends on the input (A, x)

Linear Search

```
def LS(A, x):
""" Return index i, x == A[i].
Otherwise, return -1 """
i = 0
while i < len(A):
    if A[i] == x:
        return i
    i = i + 1
return -1
```

- What is the running time complexity of $LS([2, 4, 6, 8], 4)$?
  - $A = [2, 4, 6, 8]$
  - $x = 4$
  - Number of executed lines of code =7
  - $t_{LS}([2,4,6,8], 4) = 7$

Linear Search

```
def LS(A, x):
""" Return index i, x == A[i].
Otherwise, return -1 """
i = 0
while i < len(A):
    if A[i] == x:
        return i
    i = i + 1
return -1
```

- What is the running time complexity of $LS([2, 4, 6, 8], 6)$?
  - $A = [2, 4, 6, 8]$
  - $x = 6$
  - Number of executed lines of code =10
  - $t_{LS}([2,4,6,8], 4) = 10$

Linear Search

```
def LS(A, x):
""" Return index i, x == A[i].
Otherwise, return -1 """
i = 0
while i < len(A):
    if A[i] == x:
        return i
    i = i + 1
return -1
```

- What is the running time complexity of $LS(A, x)$?
  - If the first index where x is found is k (i.e., $A[k] = x$)
  - Note: k starts from 0
  - $t_{LS}(A, x) = 1 + 3(k + 1)$
    $= 3k + 4$

Linear Search

```
def LS(A, x):
""" Return index i, x == A[i].
Otherwise, return -1 """
i = 0
while i < len(A):
    if A[i] == x:
        return i
    i = i + 1
return -1
```

- What is the running time complexity of $LS(A, x)$?
  - If x is not in A at all
  - Let n be the size of A
  - $t_{LS}(A, x) = 1 + 3n + 1 + 1$
    $= 3n + 3$

  - $t_{LS}([2,4,6,8], 99) = 15$

Takeaway

- Program running time varies with inputs
- Among inputs of a given size, there is a worst case in which the running time is the longest
- What is the worst-case running time of $LS(A, x)$ given that $\text{len}(A) = n$?
  - This is the case where x is not in A at all
  - $t_{W_{LS}}(A, x) = 3n + 3$
- Performance measures:
  - Worst-case: performance in the worst situation… interesting for this course
  - Best-case: performance in the best situation… not very interesting, rarely studied
  - Average-case: the expected performance under random inputs following certain probability distribution

Complexity of Sorting Algorithm

Insertion Sort

```
def InsSort(A) :
""" sort the elements of A in non-
decreasing order"""
1. i = 1
2. while i < len(A) :
3.     t = A[i]
4.     j = i
5.     while j > 0 and A[j-1] > t :
6.         A[j] = A[j-1]# shift up
7.         j = j-1
8.     A[j] = t
9.     i = i+1
```

- Grow a sorted list inside an unsorted list
- In each iteration:
  - Remove an element from the unsorted part
  - Insert it into the correct position in the sorted part

1	3	5	6	8	7	2	4

Worst-Case Complexity

```
def InsSort(A) :
""" sort the elements of A in non-
decreasing order"""
1. i = 1
2. while i < len(A) :
3.     t = A[i]
4.     j = i
5.     while j > 0 and A[j-1] > t :
6.         A[j] = A[j-1]# shift up
7.         j = j-1
8.     A[j] = t
9.     i = i+1
```

- Let n be the size of A
- Worst-case complexity of the algorithm:
- Worst-case j-loop:
  - $j = i, \dots, 1$
  - i iterations
  - Plus 1 line for the final loop guard (when $j \ngtr 0$)
  - In each iteration, 3 lines of code
  - Total number of lines to run: $3i + 1$
- Worst-case i-loop:
  - $i = 1, \dots, n-1$
  - $n - 1$ iterations
  - In each iteration, 5 lines of code plus the complexity of the j-loop
  - Total number of lines to run in each iteration: $(3i + 1) + 5$
  - Plus 1 line for the final loop guard (when $i \nless n$)
- Line #1

Worst-Case Complexity

```
def InsSort(A) :
""" sort the elements of A in non-
decreasing order"""
i = 1
while i < len(A) :
    t = A[i]
    j = i
    while j > 0 and A[j-1] > t :
        A[j] = A[j-1]# shift up
        j = j-1
    A[j] = t
    i = i+1
```

- $W_{t_{InsSort}}(n) = 1 + 1 + \sum_{i=1}^{n-1}\big((3i+1)+5\big)$
  $= 2 + \sum_{i=1}^{n-1}(3i+6)$
  $= 2 + \sum_{i=1}^{n-1} 3i + \sum_{i=1}^{n-1} 6$
  $= 2 + 3\sum_{i=1}^{n-1} i + 6(n-1)$
  $= 2 + 3\frac{n(n-1)}{2} + 6(n-1)$
  $= -4 + 6n + 3\frac{n(n-1)}{2}$
  $= \frac{3}{2}n^2 + \frac{9}{2}n - 4$
- Worst case complexity of insertion sort is $\frac{3}{2}n^2 + \frac{9}{2}n - 4 \in O(n^2)$

Worst-Case Complexity Examples

Example

```
def functionX(L):
""" L is a non-empty list
of length len(L) = n. """
   tot = 0
   i = 0
   while i < len(L):
      if L[i] > 0:
         tot = tot + L[i]
      i = i + 1
   return tot
```

- Worst case occurs for an all-positive input
- i-loop ($i = 0$ to $i = n - 1$):
  - Each iteration: 4 steps
  - Plus guard step
- Plus 3 steps (`tot=0, i=0, return tot`)
- Worst case complexity:

$= 1 + 3 + \sum_{i=0}^{n-1}(4) = \cdots$

$= 4n + 4$

Example

```
def functionX(L):
""" L is a non-empty list
of length len(L) = n. """
   i = 1
   while i < len(L) - 1:
      j = i - 1
      while j <= i + 1:
         L[j] = L[j] + L[i]
         j = j + 1
      i = i + 1
```

- j-loop (3 iterations):
  - Each iteration: 3 steps
  - Plus guard step
- i-loop ($i = 1$ to $i = n - 2$):
  - Each iteration: 3 steps plus j-loop
  - Plus guard step
- Plus 1 step (`i=1`)
- Worst case complexity:

$= 1 + 1 + \sum_{i=1}^{n-2}(9 + 1 + 3) = \cdots$

$= \begin{cases} 13n - 24 & n \geq 2 \\ 2 & n = 1 \end{cases}$

Example

```
def functionX(L):
  max = -10000
  i = 0
  while i < len(L):
    sum = 0
    j = i
    while j < len(L):
      sum = sum + L[j]
      if sum > max:
        max = sum
      j = j + 1
    i = i + 1
  return max
```

- Worst case occurs when (`sum>max`) is always True
- j-loop ($n - i$ iterations):
  - Each iteration: 5 steps
  - Plus guard step
- i-loop ($i = 0$ to $i = n - 1$):
  - Each iteration: 4 steps plus j-loop
  - Plus guard step
- Plus 3 steps (`max=-10000, i=0, return max`)
- Worst case complexity:

$$= 3 + 1 + \sum_{i=0}^{n-1}(5(n-i) + 1 + 4) = \cdots$$
$$= \frac{5}{2}n^2 + \frac{15}{2}n + 4$$

Example

```
def functionX(L):
""" L is a non-empty list
of length len(L) = n. """
   i = 0
   while i < len(L):
      if L[i] % 2 == 0:
         j = i
         while j < len(L):
            L[j] = L[j] + 1
            j = j + 1
      i = i + 1
```

- Worst case occurs for an "even, odd, even, odd,…" input
- j-loop ($n - i$ iterations)
  - Each iteration: 3 steps
  - Plus guard step
- i-loop ($i = 0$ to $i = n - 1$):
  - Each iteration: 4 steps plus j-loop
  - Plus guard step
- Plus 1 step (`i=0`)
- Worst case complexity:

$$= 1 + 1 + \sum_{i=0}^{n-1}(3(n-i) + 1 + 4) = \cdots$$
$$= \frac{3}{2}n^2 + \frac{13}{2}n + 2$$

Example

```
def functionX(L):
""" L is a non-empty list
of length len(L) = n. """
    i = 1
    while i < len(L):
        print L[i]
        i = i * 2
```

- Consider Line 3 (`print L[i]`)
  - If the current iteration is k (where $k \in \mathbb{N}$), then $i = 2^{k-1}$
  - At Line 4, $i = 2 \times 2^{k-1} = 2^k$
- If the kth iteration is the last one:
  - Then $k \in \mathbb{N} \wedge 2^{k-1} < n \wedge 2^k \geq n$
  - Then $k \in \mathbb{N} \wedge (k < \log_2(n) + 1) \wedge (k \geq \log_2 n)$
  - Then $k = \lceil \log_2(n) \rceil$
- Steps:
  - Each iteration has 3 steps
  - Plus loop guard step
  - Plus 1 step (`i=1`)
- Worst case complexity $= 2 + 3\lceil \log_2(n) \rceil$