Big O notation, bubble sort and insertion sort

Dr Yuhua Li
School of Computer Science & Informatics

(Content adapted from slides by Dr Louise Knight)

Outline of lecture

- Last time
- Big O notation
- Bubble sort
- Different cases
- Insertion sort

Last time

- Total comparisons = $(n - 1) + n(n - 1)/2 = n^2/2 + n/2 - 1$
- This equation has the form

  $T(n) = Cn^2 + Bn + A$

  where $T(n)$ is the runtime

- As n gets bigger and bigger, can ignore smaller terms and coefficient C and just say $T(n)$ is proportional to n^2

Big O notation

- We use “Order of Magnitude” or “Big O” notation to describe the relationship between the time, $T(n)$, taken by a particular algorithm and the number of inputs, n
- If $T(n)$ doesn’t vary with n we say it is O(1)
- If it is linear, i.e. $T(n) = An + B$ for constants A and B, we say it is $O(n)$
- If it is quadratic, i.e. $T(n) = Cn^2 + Bn + A$ for constants A, B, and C, we say it is $O(n^2)$
- (Other possibilities like $O(n^3)$, $O(\log n)$, etc.)

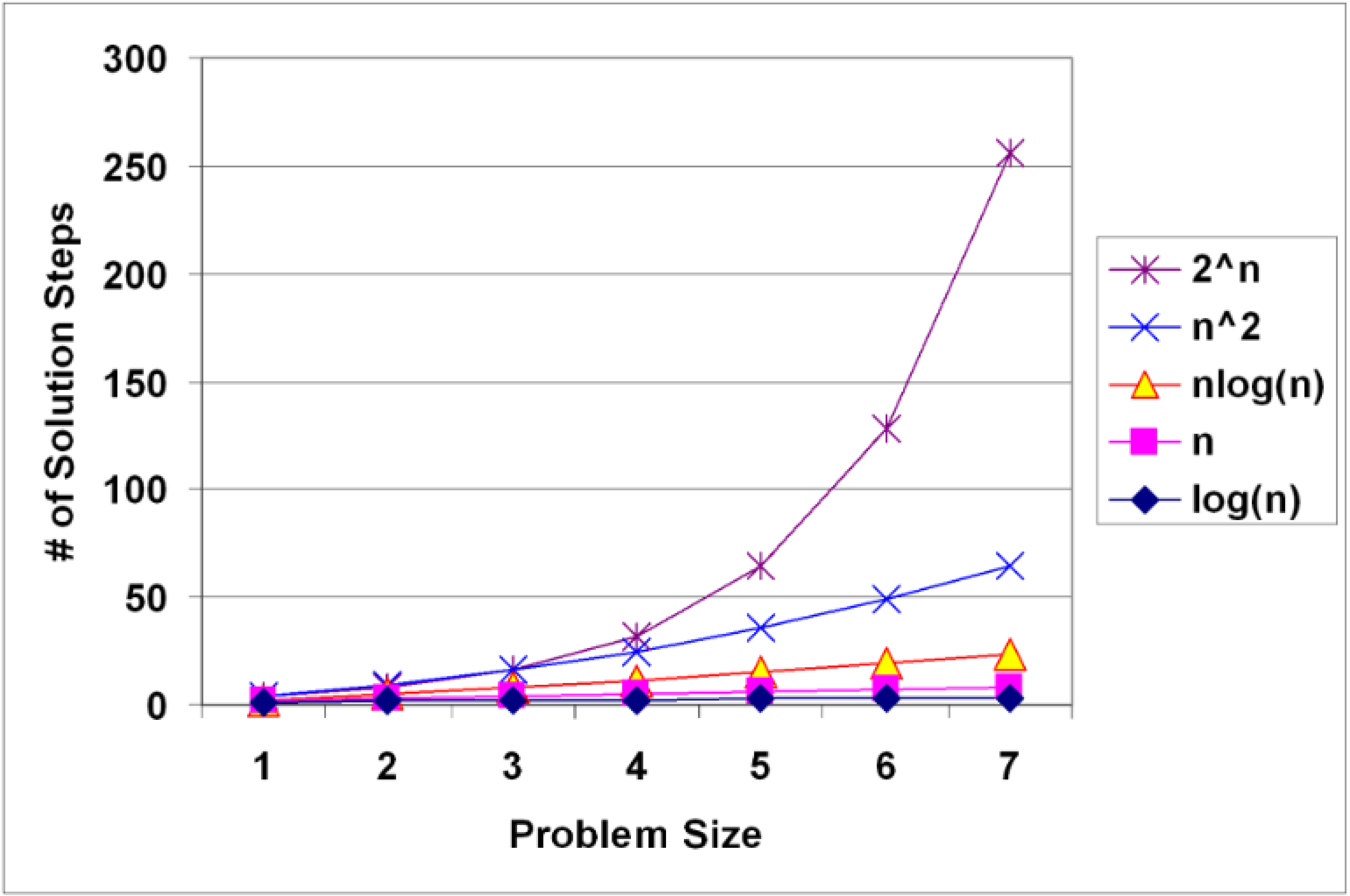
Ordering of the complexities

- Bear in mind the ordering of the complexities will influence the final big O notation

$\mathbf{1}$ $\mathbf{\log(}\boldsymbol{n}\mathbf{)}$ $\sqrt{\boldsymbol{n}}$ $\boldsymbol{n}$ $\boldsymbol{n}\,\mathbf{\log(}\boldsymbol{n}\mathbf{)}$ $\boldsymbol{n}^{\mathbf{2}}$ $\boldsymbol{n}^{\mathbf{3}}$ $\mathbf{2}^{\boldsymbol{n}}$ $\mathbf{3}^{\boldsymbol{n}}$ $\boldsymbol{n}\mathbf{!}$

Increasing complexity

Some fundamental functions

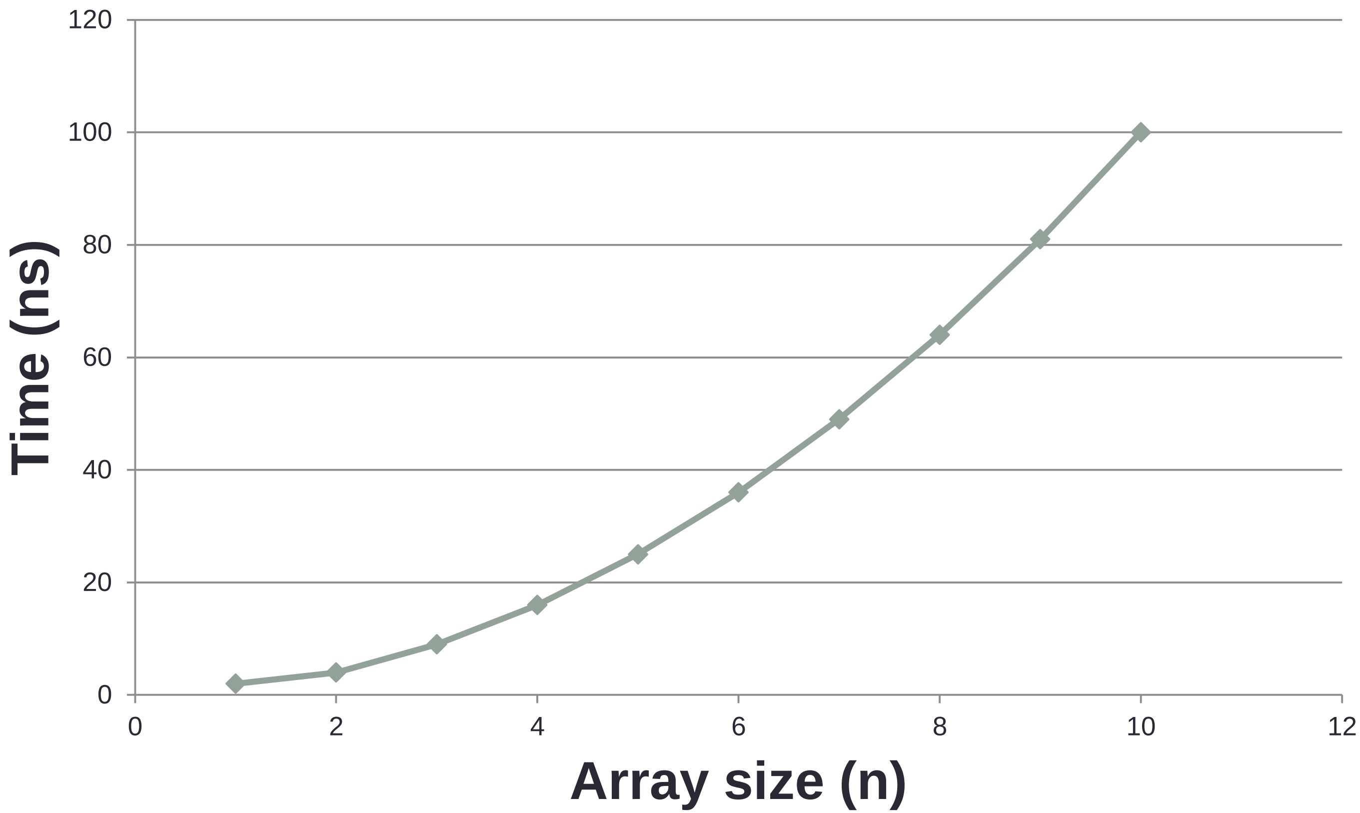


What about if we were timing code?

- Could instead plot time on y axis
- But how could we easily “prove” the complexity of an algorithm which has a theoretical complexity of n^2, for example, if we have the array size on the x axis?
- How do we know the curve is definitely n^2 and not n^3 or some other function?
- Could plot n^2 on x axis, this should give a straight line

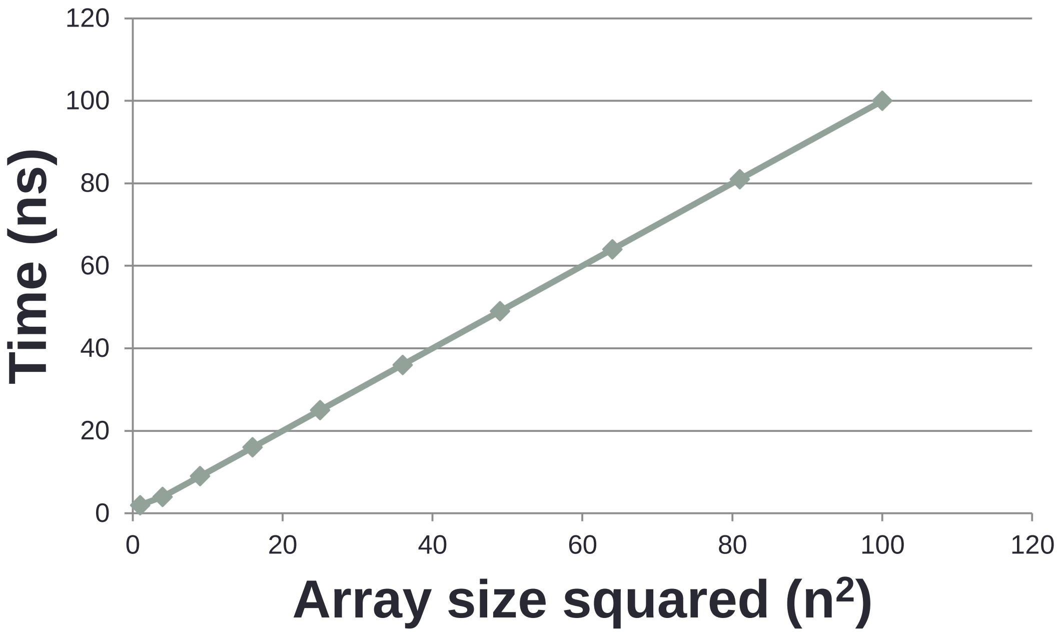
Before squaring x axis

Array size	Time (ns)
1	2
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100



After squaring x axis

Array size	Array size squared	Time (ns)
1	1	2
2	4	4
3	9	9
4	16	16
5	25	25
6	36	36
7	49	49
8	64	64
9	81	81
10	100	100



Calculating complexity

- As n increases
  - Highest complexity term dominates
  - Can ignore lower complexity terms and constants
- Quick exercise – what does each of these give in terms of big O notation?
  - $2n + 100$ gives
  - $n \log(n) + 10n$ gives
  - $\frac{1}{2} n^2 + 100n$ gives
  - $n^3 + 100n^2$ gives
  - $1/100\ 2^n + 100\ n^4$ gives

Bubble sort

1. Compare items 1 and 2 and exchange if necessary, then compare 2 and 3, 3 and 4, and so on…
2. After completing n – 1 passes of step 1, the array will be sorted

Algorithm bubble–sort(n, A)
Input: An array, A, of numbers of length n.
Output: The array, A sorted
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow 1$ **to** $n - 1$ **do**
    if $A[j+1] < A[j]$ **then**
      $temp \leftarrow A[j]$
      $A[j] \leftarrow A[j+1]$
      $A[j+1] \leftarrow temp$
    end if
  end for
end for

Bubble sort example

Input = [10, 4, 14, -3, 12, 6]

First pass:

[4, 10, 14, -3, 12, 6] Swap

[4, 10, 14, -3, 12, 6] Leave

[4, 10, -3, 14, 12, 6] Swap

[4, 10, -3, 12, 14, 6] Swap

[4, 10, -3, 12, 6, 14] Swap

Bubble sort example

Input = [10, 4, 14, -3, 12, 6]

Second pass:

[4, 10, -3, 12, 6, 14] Leave

[4, -3, 10, 12, 6, 14] Swap

[4, -3, 10, 12, 6, 14] Leave

[4, -3, 10, 6, 12, 14] Swap

[4, -3, 10, 6, 12, 14] Leave

Bubble sort example

Input = [10, 4, 14, -3, 12, 6]

Third pass:

[-3, 4, 10, 6, 12, 14] Swap

[-3, 4, 10, 6, 12, 14] Leave

[-3, 4, 6, 10, 12, 14] Swap

[-3, 4, 6, 10, 12, 14] Leave

[-3, 4, 6, 10, 12, 14] Leave

Bubble sort

- Fourth and fifth iterations have no swaps as list already sorted
- This means that time is wasted comparing elements that are already sorted
- This problem also happens with selection sort (examine all elements regardless of how sorted the array already is)
- There are algorithms where this is less of a problem, though…

Bubble sort complexity

- n – 1 comparisons per iteration (array indices start at 1 for simplicity here, would program this starting from 0)
- Two nested for loops so $(n - 1)^2$ comparisons, $O(n^2)$

Algorithm bubble–sort(n, A)
Input: An array, A, of numbers of length n.
Output: The array, A sorted
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow 1$ **to** $n - 1$ **do**
    if $A[j+1] < A[j]$ **then**
      $temp \leftarrow A[j]$
      $A[j] \leftarrow A[j+1]$
      $A[j+1] \leftarrow temp$
    end if
  end for
end for

Improved bubble sort

- Bubble sort algorithm just described is one of least efficient ways of sorting data and is rarely used
- Why is it inefficient? Comparing array elements that are already sorted (see location of 14 after first iteration)
- After i^{th} pass, the i^{th} largest element will be in correct position
- So, can reduce upper bound of inner loop by 1 for each pass

After first improvement

Algorithm bubble–sort2(n, A)
Input: An array, A, of numbers of length n.
Output: The array, A sorted
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow 1$ **to** $n - i$ **do**
    if $A[j+1] < A[j]$ **then**
      $temp \leftarrow A[j]$
      $A[j] \leftarrow A[j+1]$
      $A[j+1] \leftarrow temp$
    end if
  end for
end for

Another improvement

- Bubble sort still compares elements even when data has been completely sorted already
- Use a variable to keep track of whether any swaps took place in this iteration

Algorithm bubble–sort3(n, A)
Input: An array, A, of numbers of length n.
Output: The array, A sorted
$limit \leftarrow n - 1$
$done \leftarrow 0$
while $done = 0$ **do**
  $done \leftarrow 1$
  for $j \leftarrow 1$ **to** $limit$ **do**
    if $A[j+1] < A[j]$ **then**
      $temp \leftarrow A[j]$
      $A[j] \leftarrow A[j+1]$
      $A[j+1] \leftarrow temp$
      $done \leftarrow 0$
    end if
    $limit \leftarrow limit - 1$
  end for
end while

Different cases

- Selection sort and (naïve) bubble sort perform the same number of steps, regardless of the input
- However, there are some algorithms, like the improved bubble sort (v3) and insertion sort, that have different performances depending on the input:
  - Best case: data is already sorted e.g. [1, 2, 3, 4]
  - Average case: data is random e.g. [4, 2, 3, 1]
  - Worst case: data is reverse ordered e.g. [4, 3, 2, 1]

How do these cases apply?

- What kind of performance does each of the two algorithms we've discussed have for each of the three cases?
- Selection sort?
  - Best case
  - Average case
  - Worst case
- Improved bubble sort (v3)?
  - Best case
  - Average case $O(n^2)$
  - Worst case $O(n^2)$

Insertion sort

At every iteration, we inspect another value and find where to insert it into the already-sorted portion of the array

Algorithm insertionSort(A, n)

  Input: An array A storing n integers.

  Output: Array, A, sorted in non–descending order.

for $i = 1$ **to** $(n-1)$ **do**

    $item \leftarrow A[i]$

    $j \leftarrow i - 1$

    while $j \geq 0$ **and** $A[j] > item$ **do**

        $A[j+1] \leftarrow A[j]$

        $j \leftarrow j - 1$

    $A[j+1] \leftarrow item$

Insertion sort example

```
Algorithm insertionSort(A, n)
    Input: An array A storing n integers.
    Output: Array, A, sorted in non–descending order.
for i = 1 to (n − 1) do
        item ← A[i]
        j ← i − 1
        while j ≥ 0 and A[j] > item do
                A[j + 1] ← A[j]
                j ← j − 1
        A[j + 1] ← item
```

[10, 4, 14, -3, 12, 6]

i = 1, *item* = 4

j = 0, 10 > 4

[10, 10, 14, -3, 12, 6]

j = -1, exit while

[4, 10, 14, -3, 12, 6]

Insertion sort example

Algorithm insertionSort(A, n)
Input: An array A storing n integers.
Output: Array, A, sorted in non–descending order.
for $i = 1$ **to** $(n-1)$ **do**
    $item \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j \geq 0$ **and** $A[j] > item$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j+1] \leftarrow item$

[4, 10, 14, -3, 12, 6]

i = 2, *item* = 14

j = 1, 10 < 14, exit while

[4, 10, 14, -3, 12, 6]

Insertion sort example

Algorithm insertionSort(A, n)
Input: An array A storing n integers.
Output: Array, A, sorted in non–descending order.
for $i = 1$ **to** $(n-1)$ **do**
$item \leftarrow A[i]$
$j \leftarrow i - 1$
while $j \geq 0$ **and** $A[j] > item$ **do**
$A[j+1] \leftarrow A[j]$
$j \leftarrow j - 1$
$A[j+1] \leftarrow item$

[4, 10, 14, -3, 12, 6]

i = 3, *item* = -3

j = 2, 14 > -3

[4, 10, 14, 14, 12, 6]

j = 1, 10 > -3

[4, 10, 10, 14, 12, 6]

j = 0, 4 > -3

[4, 4, 10, 14, 12, 6]

j = -1, exit while

[-3, 4, 10, 14, 12, 6]

Insertion sort example

Algorithm insertionSort(A, n)
Input: An array A storing n integers.
Output: Array, A, sorted in non–descending order.
for $i = 1$ **to** $(n - 1)$ **do**
$item \leftarrow A[i]$
$j \leftarrow i - 1$
while $j \geq 0$ **and** $A[j] > item$ **do**
$A[j+1] \leftarrow A[j]$
$j \leftarrow j - 1$
$A[j+1] \leftarrow item$

[-3, 4, 10, 14, 12, 6]

i = 4, *item* = 12

j = 3, 14 > 12

[-3, 4, 10, 14, 14, 6]

j = 2, 10 < 12, exit while

[-3, 4, 10, 12, 14, 6]

Insertion sort example

Algorithm insertionSort(A, n)
Input: An array A storing n integers.
Output: Array, A, sorted in non–descending order.
for $i = 1$ **to** $(n - 1)$ **do**
$item \leftarrow A[i]$
$j \leftarrow i - 1$
while $j \geq 0$ **and** $A[j] > item$ **do**
$A[j+1] \leftarrow A[j]$
$j \leftarrow j - 1$
$A[j+1] \leftarrow item$

[-3, 4, 10, 12, 14, 6]

i = 5, *item* = 6

j = 4, 14 > 6

[-3, 4, 10, 12, 14, 14]

j = 3, 12 > 6

[-3, 4, 10, 12, 12, 14]

j = 2, 10 > 6

[-3, 4, 10, 10, 12, 14]

j = 1, 4 < 6, exit while

[-3, 4, 6, 10, 12, 14]

Different cases for insertion sort

- O(n) best case – already sorted [1, 2, 3, 4]
- O(n^2) average case – random array [2, 1, 4, 3]
- O(n^2) worst case – reverse sorted array [4, 3, 2, 1]