CSE21 - Math for Algorithm and Systems Analysis
Asymptotic Analysis : Building Better Algorithms

Russell Impagliazzo and Miles Jones. Thanks to Janine Tiefenbruck

April 6, 2016

Today's agenda

1. Using order notation to analyse algorithm time: some simple rules
2. Finding better algorithms: When is an improvement meaningful?
3. Illustrate some basic algorithm design principles: pre-processing, re-use of computation

Order of time taken by algorithms

Order is frequently used to describe the time taken by algorithms. We want a simple expression that estimates the time $T(n)$ the algorithm takes on an input of size n. (This can be the worst-case time, best-case time or average time. Most frequently, it is worst-case, because that is the most useful to know.)

Basic operation: A basic operation is one whose time *does not depend on the input.* Because of this, any basic operation takes *constant* time, $O(1)$. (Each one is some fixed number, which might depend on all the factors we discussed last class, and order does not distinguish between different constants).

Order of time for algorithms, cont.

Simple loops: If the guards of a loop are basic operations, and the body is constant time, the time the loop takes will be of the same order as the *total number of iterations*.

Combining non-nested parts: The time to do two separate non-nested algorithms is the sum of the times for them individually. By the additivity property, the order of a sum is the maximum. So for two non-nested parts of an algorithm, the time for the whole is the *greater* of the parts.

Analyzing nested loops: simple case

Suppose a loop will be executed at most T_1 times, and each time, the body (the inner loop) gets executed. If we've already analysed the body as taking time $O(T_2)$ in the worst case, we can conclude that the total time for the loop is *the product of the number of iterations and the time for the body*, i.e., $O(T_1 * T_2)$.

Remember that O is an *upper bound*, not the exact amount of time. Sometimes, this bound is not *tight*, i.e., there are smaller upper bounds that are also true. We will talk about this next class.

Sub-routines

If we use a sub-routine S in our algorithm, and we have already analysed its time as $T_S(n)$, then the total cost for all invocations of the sub-routine will be *at most* the number of times we use it times the worst-case time it might take. So if we use it T_1 times (such as in a loop with T_1 executions), and we use it on inputs of size at most m, the total time for all uses is $O(T_1 T_S(m))$.

Note that we need to distinguish m the size of input we feed to the sub-routine from n, the original input size for the main procedure.

Example: Selection (Min) Sort

$MinSort(A[1, \ldots, n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n - 1$ do:
2. $\quad Min \leftarrow A[K], index \leftarrow K$
3. $\quad$ FOR $J \leftarrow K + 1$ TO n do:
4. $\quad\quad$ IF $A[J] < Min$ THEN $Min \leftarrow A[J], index \leftarrow J$.
5. $\quad A[index] \leftarrow A[K], A[K] \leftarrow Min$.

We work from the inside out, going from the body of the inside loop to the main algorithm.

Example: Selection (Min) Sort

$MinSort(A[1,\ldots,n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. $\quad Min \leftarrow A[K], index \leftarrow K$
3. $\quad$ FOR $J \leftarrow K+1$ TO n do:
4. $\qquad$ IF $A[J] < Min$ THEN $Min \leftarrow A[J], index \leftarrow J$.
5. $\quad A[index] \leftarrow A[K], A[K] \leftarrow Min$.

The inner-most Line 4 is defined in terms of a fixed number of basic operations: a comparison, some logic, some variable writes. It is thus $O(1)$.

Example: Selection (Min) Sort

$MinSort(A[1,\ldots,n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. $\quad Min \leftarrow A[K], index \leftarrow K$
3. $\quad$ FOR $J \leftarrow K+1$ TO n do:
4. $\quad\quad O(1)$ time
5. $\quad A[index] \leftarrow A[K], A[K] \leftarrow Min.$

Example: Selection (Min) Sort

$MinSort(A[1,\ldots,n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. $\quad Min \leftarrow A[K], index \leftarrow K$
3. $\quad$ FOR $J \leftarrow K+1$ TO n do:
4. $\quad\quad O(1)$ time
5. $\quad A[index] \leftarrow A[K], A[K] \leftarrow Min.$

Example: Selection (Min) Sort

$MinSort(A[1,\ldots,n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. $\quad Min \leftarrow A[K], index \leftarrow K$
3. $\quad$ FOR $J \leftarrow K+1$ TO n do:
4. $\quad\quad O(1)$ time
5. $\quad A[index] \leftarrow A[K], A[K] \leftarrow Min.$

Line 3 is a loop, with constant time line 4 inside. It repeats $n-K$ times, so the total time is $O(n-K)$. This ranges from constant time when K reaches $n-1$ to $O(n)$ when $K=1$. So the worst-case is $O(n)$.

Example: Selection (Min) Sort

$MinSort(A[1,\ldots,n])$: $arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. $Min \leftarrow A[K], index \leftarrow K$
3. $O(n)$ time loop
4. already included
5. $A[index] \leftarrow A[K], A[K] \leftarrow Min.$

Example: Selection (Min) Sort

$MinSort(A[1,\ldots,n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. $Min \leftarrow A[K], index \leftarrow K$
3. $O(n)$ time loop
4. already included
5. $A[index] \leftarrow A[K], A[K] \leftarrow Min.$

Example: Selection (Min) Sort

$MinSort(A[1, \dots, n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n - 1$ do:
2. $\quad Min \leftarrow A[K], index \leftarrow K$
3. $\quad O(n)$ time loop
4. $\quad$ already included
5. $\quad A[index] \leftarrow A[K], A[K] \leftarrow Min.$

Example: Selection (Min) Sort

$MinSort(A[1,\ldots,n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. $\quad Min \leftarrow A[K], index \leftarrow K$
3. $\quad O(n)$ time loop
4. $\quad$ already included
5. $\quad A[index] \leftarrow A[K], A[K] \leftarrow Min.$

Line 2 and 5 are constant time, so the body of the FOR loop in line 1 takes $O(1+n+1) = O(n)$ total.

Example: Selection (Min) Sort

$MinSort(A[1,\ldots,n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. $O(1)$ time
3. $O(n)$ time loop
4. already included
5. $O(1)$ time

Example: Selection (Min) Sort

$MinSort(A[1, \ldots, n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n - 1$ do:
2. $O(1)$ time
3. $O(n)$ time loop
4. already included
5. $O(1)$ time

Example: Selection (Min) Sort

$MinSort(A[1,\ldots,n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. $O(1)$ time
3. $O(n)$ time loop
4. already included
5. $O(1)$ time

Example: Selection (Min) Sort

$MinSort(A[1, \ldots, n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. $O(1)$ time
3. $O(n)$ time loop
4. already included
5. $O(1)$ time

Line 2 and 5 are constant time, so the body of the FOR loop in line 1 takes $O(1+n+1) = O(n)$ total.

Example: Selection (Min) Sort

$MinSort(A[1,\ldots,n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. absorbed below
3. $O(n)$ time
4. already included
5. absorbed above

Finally, line 1 is a loop whose body is $O(n)$ and gets repeated $n-1 < n$ times So the whole algorithm is $O(n^2)$.

$MinSort(A[1,\ldots,n]) : arrayofintegers$

1. $O(n^2)$ time
2. included
3. included
4. included
5. included

Is this the best answer?

O is an upper bound, not always tight. We can ask: is the running time also lower bounded by a quardratic, or is there a smaller upper bound? We don't need to find the "worst-case input" or give an exact formula to answer this question, just show that sometimes the algorithm performs at least on the order of n^2 operations of some kind.

$MinSort(A[1..n]) : arrayofintegers$

1. FOR $K \leftarrow 1$ TO $n-1$ do:
2. $\quad Min \leftarrow A[K], index \leftarrow K$
3. $\quad$ FOR $J \leftarrow K+1$ TO n do:
4. $\quad\quad$ IF $A[J] < Min$ THEN $Min \leftarrow A[J], index \leftarrow J$.
5. $\quad A[index] \leftarrow A[K], A[K] \leftarrow Min$.

Look at the first $n/2$ times we run the loop in line 3. Then $K \leq n/2$, so $n - K \geq n/2$. Thus, we run it at least $n/2 * n/2 = n^2/4$ times total. This is $\Omega(n^2)$. Thus, the time is both $O(n^2)$ and $\Omega(n^2)$, so our analysis is tight, and the time is $\Theta(n^2)$. So in this example, our first analysis is the best possible.

Order counts all operations

Here's an example where the non-comparison operations for a sorting algorithm dominate run time. So counting just the comparisons doesn't tell us the complete picture.
$BinaryInsertSortr(A[1..n] : arrayofintegers)$

1. FOR $K \leftarrow 2$ to n do:
2. Use bin. search to find predecessor position p of $A[K]$ in $A[1...K-1]$.
3. Save $A[K]$ as V
4. Move elements $p+1...K-1$ over one place in the array.
5. $A[p+1] \leftarrow V$
6. Return $A[1..n]$.

Note that we use at most $\log n$ comparisons to perform the binary search in line 2, and the other operations don't involve comparisons at all. So the total number of comparisons is at most $n \log n$.

Order counts all operations

Using our standard inside-out method:
$BinaryInsertSortr(A[1..n] : arrayofintegers)$

1. FOR $K \leftarrow 2$ to n do:
2. binary search: $O(\log n)$ time Use bin. search to find predecessor position p of $A[K]$ in $A[1...K-1]$.
3. $O(1)$ time
4. Up to n elements to move $= O(n)$ time
5. $O(1)$ time
6. Return $A[1..n]$.

So the total time for the inside of the loop (lines 2-5) is:

A $O(\log n)$

B $O(1)$

C $O(n)$

D $O(n^2)$

E None of the above

Order counts all operations

Using our standard inside-out method: *BinaryInsertSort*$(A[1..n])$

1. FOR $K \leftarrow 2$ to n do:
2. Total time $O(n)$
3. Return $A[1..n]$.

Hence, total time over all is $O(n^2)$.

Is this tight ?

BinaryInsertSortr$(A[1..n] : arrayofintegers)$

1. FOR $K \leftarrow 2$ to n do:
2. Use bin. search to find predecessor position p of $A[K]$ in $A[1...K-1]$.
3. Save $A[K]$ as V
4. Move elements $p+1...K-1$ over one place in the array.
5. $A[p+1] \leftarrow V$
6. Return $A[1..n]$.

Which is true?

- A The time is $\Omega(n^2)$ on an already sorted input
- B The time is $\Omega(n^2)$ on a reversely sorted input
- C Both of the above
- D The time is never $\Omega(n^2)$.

How O distinguishes between major and incremental improvements

We have already seen the defininition of O and related order notations, and have seen some simple ways of using the properties of O to analyze the time of algorithms up to order.

The summing triple problem

- Input: An array $A[1, \ldots, n]$ of integers.
- Summing Triple: A **summing triple** is a list of three indices $1 \leq I, J, K \leq n$ so that $A[I] + A[J] = A[K]$.
- Problem: Is there a summing triple?
- Example: If $A[1..5] = [3, 6, 5, 7, 8]$, $1, 3, 5$ would be a summing triple, since $A[1] + A[3] = A[5]$.

Most Obvious Algorithm

SumTriples($A[1, \ldots, n]$)

1. FOR $I = 1$ TO n do:
2. FOR $J = 1$ TO n do:
3. FOR $K = 1$ to n do:
4. IF $A[I] + A[J] = A[K]$ THEN Return *True*
5. Return *False*

This algorithm's time is

- A $O(n)$
- B $O(n^2 \log n)$
- C $O(n^2)$
- D $O(n^3)$

Time analysis worked out

SumTriples($A[1, \ldots, n]$)

```
1. FOR I = 1 TO n do:
2.     FOR J = 1 TO n do:
3.         FOR K = 1 to n do:
4.             IF A[I] + A[J] = A[K] THEN Return True
5. Return False
```

Time analysis: Line 4 : $O(1)$ time.
Three nested loops each always make n iterations, so n^3 total iterations.
Therefore, $T(n) \in O(n^3)$.

Eliminating Some Redundancy

SumTriples($A[1, \ldots, n]$)

1. FOR $I = 1$ TO n do:
2. FOR $J = I$ TO n do:
3. FOR $K = 1$ to n do:
4. IF $A[I] + A[J] = A[K]$ THEN Return *True*
5. Return *False*

Before, we checked every I and J twice, in both orders. So this algorithm has eliminated about half of the work of the previous one. But a constant factor of $1/2$ does not change the order, so $T(n) \in O(n^3)$ still.

Viewing the algorithm more conceptually

Here's another way of describing the same algorithm:
For each $1 \leq I \leq J \leq n$, we use *linear search* to see if $A[I] + A[J]$ is in the array $A[1, \ldots, n]$.

It doesn't change the algorithm, but it raises the possibility of using a *different search* to replace linear search.

If the array were sorted

If we knew A was sorted, then we could replace the linear search with *binary search*.

SortedSumTriples($A[1, \ldots, n]$: sorted array of integers)

1. FOR $I = 1$ TO n do:
2. FOR $J = I$ TO n do:
3. IF $BinarySearch(A, A[I] + A[J])$ Then Return *True*
4. Return *False*

How long would this take?

- A $O(n)$
- B $O(n^2 \log n)$
- C $O(n^2)$
- D $O(n^3)$

Time analysis of SortedSumTriples

SortedSumTriples($A[1, \ldots, n]$: sorted array of integers)

1. FOR $I = 1$ TO n do:
2. FOR $J = I$ TO n do:
3. IF $BinarySearch(A, A[I] + A[J])$ Then Return *True*
4. Return *False*

Since binary search takes $O(\log n)$ time, and we have two nested loops with fewer than n iterations each, the total time is $O(n^2 \log n)$.

No assumptions

We cannot **assume** the array A is sorted, but we can **ensure** that it is sorted:

SumTriples($A[1, \ldots, n]$: array of integers)

1. *BubbleSort*(A)
2. Return *SortedSumTriples*(A).

Is it better?

How much time does SumTriples take? Is it better or worse than our first $O(n^3)$ algorithm?

SumTriples($A[1, \ldots, n]$: array of integers)

1. *BubbleSort*(A)
2. Return *SortedSumTriples*(A).

Analysis of new algorithm

The new algorithm has two unnested parts, sorting and then using Sorted Sum Triples. We've already analyzed the two parts. BubbleSort takes time $O(n^2)$, and SortedSumTriples takes time $O(n^2 \log n)$. So the total time is $O(n^2 + n^2 \log n) = O(n^2 \log n)$.

Because BubbleSort's time is o of the total time, a better sorting procedure won't improve the total time significantly.

Because $n^2 \log n \in o(n^3)$, this is an asymptotically strictly better algorithm than what we started with.

Digression

The best algorithms known for SumTriple take $O(n^2)$ time. The question of whether there is a better algorithm than that is unknown, and the subject of much active research. If *SumTriples* really does require about $O(n^2$ time so do many other problems in geometry, such as testing whether points are in "general position",i..e., no three co-linear.