Introduction to Algorithms and Data Structures

Lecture 5: Asymptotics for Insertsort and Mergesort

John Longley

School of Informatics
University of Edinburgh

29 September 2025

Algorithms and cost models

We're interested in the cost of various algorithms, for various notions of cost. (Runtime, memory use, disk operations, ...).

Algorithms and cost models

We're interested in the cost of various algorithms, for various notions of cost. (Runtime, memory use, disk operations, ...).

To analyse this, need some cost model: i.e. some definition of how we intend cost to be measured.
Different cost models are useful for different purposes.

Algorithms and cost models

We're interested in the cost of various algorithms, for various notions of cost. (Runtime, memory use, disk operations, . . .).

To analyse this, need some cost model: i.e. some definition of how we intend cost to be measured.
Different cost models are useful for different purposes.

We'll initially consider runtime cost. But even here, different cost models are possible. E.g. for sorting algorithms, might measure. . .

Algorithms and cost models

We're interested in the cost of various algorithms, for various notions of cost. (Runtime, memory use, disk operations, ...).

To analyse this, need some cost model: i.e. some definition of how we intend cost to be measured.
Different cost models are useful for different purposes.

We'll initially consider runtime cost. But even here, different cost models are possible. E.g. for sorting algorithms, might measure...

1. number of comparisons ($<$) between list items,

Algorithms and cost models

We're interested in the cost of various algorithms, for various notions of cost. (Runtime, memory use, disk operations, ...).

To analyse this, need some cost model: i.e. some definition of how we intend cost to be measured.
Different cost models are useful for different purposes.

We'll initially consider runtime cost. But even here, different cost models are possible. E.g. for sorting algorithms, might measure...

1. number of comparisons ($<$) between list items,
2. number of 'basic steps' performed – e.g. 'machine instructions' for some (idealized) machine model.

Algorithms and cost models

We're interested in the cost of various algorithms, for various notions of cost. (Runtime, memory use, disk operations, ...).

To analyse this, need some cost model: i.e. some definition of how we intend cost to be measured.
Different cost models are useful for different purposes.

We'll initially consider runtime cost. But even here, different cost models are possible. E.g. for sorting algorithms, might measure...

1. number of comparisons ($<$) between list items,
2. number of 'basic steps' performed – e.g. 'machine instructions' for some (idealized) machine model.

This lecture: Start with 1, then move towards 2.

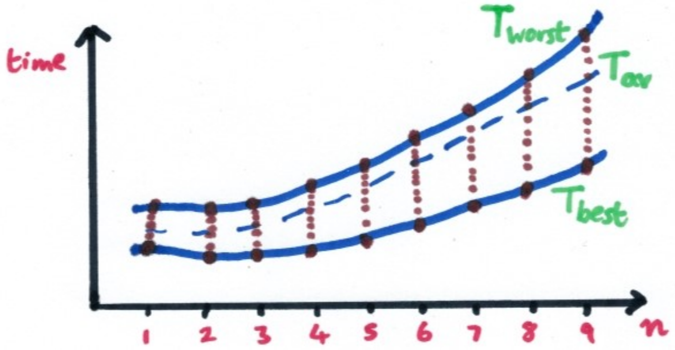
Best case, worst case, average case

Often want to estimate runtime on 'inputs of size n'.
(E.g. time taken to sort lists of length n.)

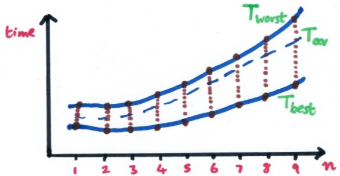
Best case, worst case, average case

Often want to estimate runtime on 'inputs of size n'.
(E.g. time taken to sort lists of length n.)

Typically, different inputs of size n give different runtimes!
However, the best case, worst case and (sometimes) average case times give well-defined functions we can talk about.

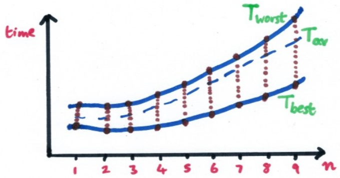


Best/worst case and asymptotic bounds



Informally . . .

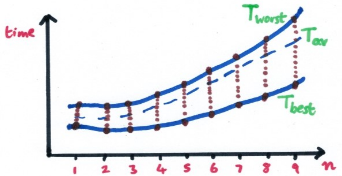
Best/worst case and asymptotic bounds



Informally . . .

- $T_{worst} = O(g)$ says the worst-case runtime (hence any runtime) is essentially no worse than g. Can then just say 'runtime is $O(g)$'.

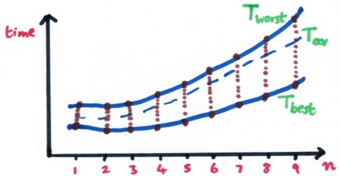
Best/worst case and asymptotic bounds



Informally ...

- $T_{worst} = O(g)$ says the worst-case runtime (hence any runtime) is essentially no worse than g. Can then just say 'runtime is $O(g)$'.
- $T_{worst} = \Omega(g)$ says runtime can be as bad as g, i.e. there *are* inputs that manifest this bad behaviour.

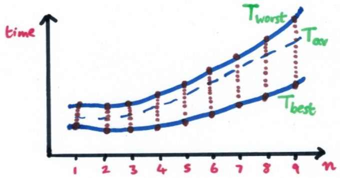
Best/worst case and asymptotic bounds



Informally ...

- $T_{worst} = O(g)$ says the worst-case runtime (hence any runtime) is essentially no worse than g. Can then just say 'runtime is $O(g)$'.
- $T_{worst} = \Omega(g)$ says runtime can be as bad as g, i.e. there *are* inputs that manifest this bad behaviour.
- $T_{worst} = \Theta(g)$ says both these things.

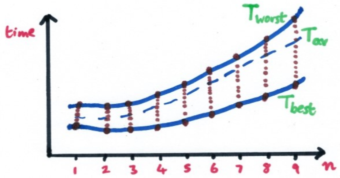
Best/worst case and asymptotic bounds



Informally ...

- $T_{worst} = O(g)$ says the worst-case runtime (hence any runtime) is essentially no worse than g. Can then just say 'runtime is $O(g)$'.
- $T_{worst} = \Omega(g)$ says runtime can be as bad as g, i.e. there *are* inputs that manifest this bad behaviour.
- $T_{worst} = \Theta(g)$ says both these things.
- $T_{best} = \Omega(g)$ says best-case runtime (hence any runtime) is essentially no better than g. Can then say 'runtime is $\Omega(g)$'.
- $T_{best} = O(g)$ says runtime can be as good as g.
- $T_{best} = \Theta(g)$ says both these things.

Best/worst case and asymptotic bounds



Informally . . .

- $T_{worst} = O(g)$ says the worst-case runtime (hence any runtime) is essentially no worse than g. Can then just say 'runtime is $O(g)$'.
- $T_{worst} = \Omega(g)$ says runtime can be as bad as g, i.e. there *are* inputs that manifest this bad behaviour.
- $T_{worst} = \Theta(g)$ says both these things.
- $T_{best} = \Omega(g)$ says best-case runtime (hence any runtime) is essentially no better than g. Can then say 'runtime is $\Omega(g)$'.
- $T_{best} = O(g)$ says runtime can be as good as g.
- $T_{best} = \Theta(g)$ says both these things.

Moral: Don't confuse '$O/\Omega/\Theta$' with 'best/worst/average'!
(See CLRS pages 48-49.)

InsertSort: ‘number of comparisons’ analysis

Pseudocode for InsertSort again (line numbers added):

```
0  InsertSort(A):
1      for i = 1 to n−1        # write n for size of A
2          x = A[i]
3          j = i−1
4          while j ≥ 0 and A[j] > x
5              A[j+1] = A[j]
6              j = j−1
7          A[j+1] = x
```

How many times is the ‘>’ on line 4 invoked?

InsertSort: 'number of comparisons' analysis

Pseudocode for InsertSort again (line numbers added):

```
0  InsertSort(A):
1      for i = 1 to n−1        # write n for size of A
2          x = A[i]
3          j = i−1
4          while j ≥ 0 and A[j] > x
5              A[j+1] = A[j]
6              j = j−1
7          A[j+1] = x
```

How many times is the '$>$' on line 4 invoked?

- For each value of i, may consider execution of lines 2–7. This invokes $>$ at most i times.
  (Loop starts at j = i−1 and stops at j = −1, if not before).

InsertSort: ‘number of comparisons’ analysis

Pseudocode for InsertSort again (line numbers added):

```
0  InsertSort(A):
1     for i = 1 to n−1        # write n for size of A
2         x = A[i]
3         j = i−1
4         while j ≥ 0 and A[j] > x
5             A[j+1] = A[j]
6             j = j−1
7         A[j+1] = x
```

How many times is the ‘$>$’ on line 4 invoked?

- For each value of i, may consider execution of lines 2–7. This invokes $>$ at most i times.
  (Loop starts at $j = i-1$ and stops at $j = -1$, if not before).
- But i itself runs from 1 to $n-1$ (line 1).

InsertSort: 'number of comparisons' analysis

Pseudocode for InsertSort again (line numbers added):

```
0 InsertSort(A):
1     for i = 1 to n−1        # write n for size of A
2         x = A[i]
3         j = i−1
4         while j ≥ 0 and A[j] > x
5             A[j+1] = A[j]
6             j = j−1
7         A[j+1] = x
```

How many times is the '$>$' on line 4 invoked?

- For each value of i, may consider execution of lines 2–7. This invokes $>$ at most i times.
  (Loop starts at j = i−1 and stops at j = −1, if not before).
- But i itself runs from 1 to n−1 (line 1).
- So total number of '$>$' ops is at most $\Sigma_{i=1}^{n-1} i = O(n^2)$.

'Worst case' number of comparisons

We've seen that $\sum_{i=1}^{n-1} i = n(n-1)/2$ is an upper bound for number of comparisons. So can say InsertSort does $O(n^2)$ comparisons.

'Worst case' number of comparisons

We've seen that $\sum_{i=1}^{n-1} i = n(n-1)/2$ is an upper bound for number of comparisons. So can say InsertSort does $O(n^2)$ comparisons.

But is it ever actually this bad? I.e. is this upper bound attained?

‘Worst case’ number of comparisons

We’ve seen that $\sum_{i=1}^{n-1} i = n(n-1)/2$ is an upper bound for number of comparisons. So can say InsertSort does $O(n^2)$ comparisons.

But is it ever actually this bad? I.e. is this upper bound attained?

For any n, consider how InsertSort will behave on the input $[n, n-1, \ldots, 2, 1]$. Recall the inner loop:

```
while j ≥ 0 and A[j] > x
    A[j+1] = A[j]
    j = j−1
```

'Worst case' number of comparisons

We've seen that $\sum_{i=1}^{n-1} i = n(n-1)/2$ is an upper bound for number of comparisons. So can say InsertSort does $O(n^2)$ comparisons.

But is it ever actually this bad? I.e. is this upper bound attained?

For any n, consider how InsertSort will behave on the input $[n, n-1, \ldots, 2, 1]$. Recall the inner loop:

```
while j ≥ 0 and A[j] > x
    A[j+1] = A[j]
    j = j–1
```

Not hard to see that *all* the comparisons A[j] $>$ x will yield True.
So for each i, this 'j-loop' will run until j $= -1$.
So there will be exactly $n(n-1)/2$ comparisons.

‘Worst case’ number of comparisons

We’ve seen that $\Sigma_{i=1}^{n-1} i = n(n-1)/2$ is an upper bound for number of comparisons. So can say InsertSort does $O(n^2)$ comparisons.

But is it ever actually this bad? I.e. is this upper bound attained?

For any n, consider how InsertSort will behave on the input $[n, n-1, \ldots, 2, 1]$. Recall the inner loop:

```
while j ≥ 0 and A[j] > x
    A[j+1] = A[j]
    j = j−1
```

Not hard to see that *all* the comparisons A[j] $>$ x will yield True.
So for each i, this ‘j-loop’ will run until j $= -1$.
So there will be exactly $n(n-1)/2$ comparisons.

Headline is that the worst-case time is $\Omega(n^2)$, hence $\Theta(n^2)$.

What about 'best case'?

What's the smallest number of comparisons that InsertSort could possibly perform on a size n input?

What about 'best case'?

What's the smallest number of comparisons that InsertSort could possibly perform on a size n input?

```
j = i−1
while j ≥ 0 and A[j] > x
      A[j+1] = A[j]
      j = j−1
```

For each value of i, this j-loop will do at least one comparison (first time round, when j = i−1).
So the whole program performs at least n−1 comparisons.

What about ‘best case’?

What’s the smallest number of comparisons that InsertSort could possibly perform on a size n input?

```
j = i−1
while j ≥ 0 and A[j] > x
    A[j+1] = A[j]
    j = j−1
```

For each value of i, this j-loop will do at least one comparison (first time round, when j = i−1).
So the whole program performs at least n−1 comparisons.

Is this lower bound attained? Yes: when input is already sorted!
So best-case number of comparisons is $\Theta(n)$.

What about 'best case'?

What's the smallest number of comparisons that InsertSort could possibly perform on a size n input?

```
j = i−1
while j ≥ 0 and A[j] > x
    A[j+1] = A[j]
    j = j−1
```

For each value of i, this j-loop will do at least one comparison (first time round, when j = i−1).
So the whole program performs at least n−1 comparisons.

Is this lower bound attained? Yes: when input is already sorted!
So best-case number of comparisons is $\Theta(n)$.

Can now say: number of comparisons performed by InsertSort on size n inputs is $O(n^2)$ and $\Omega(n)$.

'Average case' for InsertSort (brief glimpse)

We've looked at worst and best cases. But how many comparisons will InsertSort perform on average?

'Average case' for InsertSort (brief glimpse)

We've looked at worst and best cases. But how many comparisons will InsertSort perform on average?

For simplicity, assume input A is some permutation of n *distinct* elements $x_0 < \ldots < x_{n-1}$, with all $n!$ permutations 'equally likely'. Let $T_{av}(n) =$ average number of comparisons for these $n!$ inputs.

‘Average case’ for InsertSort (brief glimpse)

We’ve looked at worst and best cases. But how many comparisons will InsertSort perform on average?

For simplicity, assume input A is some permutation of n *distinct* elements $x_0 < \ldots < x_{n-1}$, with all $n!$ permutations ‘equally likely’. Let $T_{av}(n) =$ average number of comparisons for these $n!$ inputs.

That is, if P_n is the set of $n!$ orderings of $\{x_0, x_1, \ldots, x_{n-1}\}$, then

$$T_{av}(n) = \frac{1}{n!} \sum_{p \in P_n} (\sharp \text{ comparisons on input } [p(0), \ldots, p(n-1)])$$

'Average case' for InsertSort (brief glimpse)

We've looked at worst and best cases. But how many comparisons will InsertSort perform on average?

For simplicity, assume input A is some permutation of n *distinct* elements $x_0 < \ldots < x_{n-1}$, with all $n!$ permutations 'equally likely'. Let $T_{av}(n) =$ average number of comparisons for these $n!$ inputs.

That is, if P_n is the set of $n!$ orderings of $\{x_0, x_1, \ldots, x_{n-1}\}$, then

$$T_{av}(n) = \frac{1}{n!} \sum_{p \in P_n} (\sharp \text{ comparisons on input } [p(0), \ldots, p(n-1)])$$

It can be shown that

$$T_{av}(n) = n^2/4 - O(n)$$

(compare $T_{worst}(n) = n^2/2 - O(n)$). Anyway, $T_{av}(n) = \Theta(n^2)$.

MergeSort: comparison analysis

First recall the **Merge** operation for merging two already sorted arrays B and C, of combined length m (i.e. $|B| + |C| = m$).

MergeSort: comparison analysis

First recall the **Merge** operation for merging two already sorted arrays B and C, of combined length m (i.e. $|B| + |C| = m$).

How many comparisons does this perform?
(Assume we stop doing comparisons once B or C is used up.)

MergeSort: comparison analysis

First recall the **Merge** operation for merging two already sorted arrays B and C, of combined length m (i.e. $|B| + |C| = m$).

How many comparisons does this perform?
(Assume we stop doing comparisons once B or C is used up.)

Reasoning informally ...

- At most $m - 1$. Every comparison yields a new element for the result list D, and the very last element gets put in without a comparison.

MergeSort: comparison analysis

First recall the **Merge** operation for merging two already sorted arrays B and C, of combined length m (i.e. $|B| + |C| = m$).

How many comparisons does this perform?
(Assume we stop doing comparisons once B or C is used up.)

Reasoning informally ...

- At most $m - 1$. Every comparison yields a new element for the result list D, and the very last element gets put in without a comparison.
- At least $\min(|B|, |C|)$. And if $|B|$ and $|C|$ differ by at most 1 (as they will in MergeSort), this is $m/2 - O(1)$.

MergeSort: comparison analysis

First recall the **Merge** operation for merging two already sorted arrays B and C, of combined length m (i.e. $|B| + |C| = m$).

How many comparisons does this perform?
(Assume we stop doing comparisons once B or C is used up.)

Reasoning informally ...

- At most $m - 1$. Every comparison yields a new element for the result list D, and the very last element gets put in without a comparison.
- At least $\min(|B|, |C|)$. And if $|B|$ and $|C|$ differ by at most 1 (as they will in MergeSort), this is $m/2 - O(1)$.

So can say that number of comparisons done by **Merge** (within **MergeSort**) is $\Theta(m)$.

MergeSort: comparison analysis

First recall the **Merge** operation for merging two already sorted arrays B and C, of combined length m (i.e. $|B| + |C| = m$).

How many comparisons does this perform?
(Assume we stop doing comparisons once B or C is used up.)

Reasoning informally ...

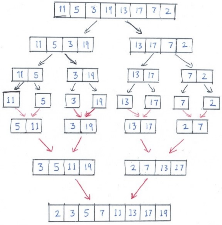
- At most $m - 1$. Every comparison yields a new element for the result list D, and the very last element gets put in without a comparison.
- At least $\min(|B|, |C|)$. And if $|B|$ and $|C|$ differ by at most 1 (as they will in MergeSort), this is $m/2 - O(1)$.

So can say that number of comparisons done by **Merge** (within **MergeSort**) is $\Theta(m)$.

Now what about **MergeSort** itself?

Analysis of MergeSort, ctd.

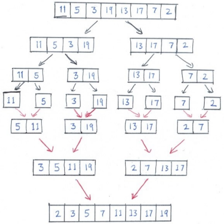
For simplicity, suppose first that we're sorting a list of size $n = 2^k$.
We'll reason informally. Recall our diagram:



All the comparisons/merging happen in the bottom half.

Analysis of MergeSort, ctd.

For simplicity, suppose first that we're sorting a list of size $n = 2^k$. We'll reason informally. Recall our diagram:

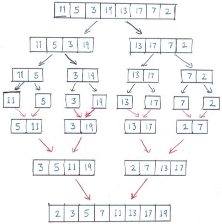


All the comparisons/merging happen in the bottom half.

- On each 'level', total number of elements is n.

Analysis of MergeSort, ctd.

For simplicity, suppose first that we're sorting a list of size $n = 2^k$.
We'll reason informally. Recall our diagram:

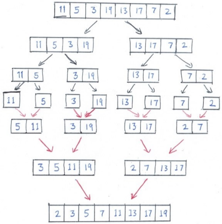


All the comparisons/merging happen in the bottom half.

- On each 'level', total number of elements is n.
- And for each merge, $\sharp$comparisons $<$ $\sharp$elements involved.

Analysis of MergeSort, ctd.

For simplicity, suppose first that we're sorting a list of size $n = 2^k$.
We'll reason informally. Recall our diagram:

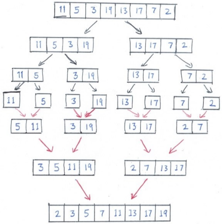


All the comparisons/merging happen in the bottom half.

- On each 'level', total number of elements is n.
- And for each merge, $\sharp$comparisons $<$ $\sharp$elements involved.
- So total $\sharp$comparisons for merges on each level is $< n$.

Analysis of MergeSort, ctd.

For simplicity, suppose first that we're sorting a list of size $n = 2^k$.
We'll reason informally. Recall our diagram:

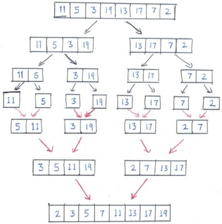


All the comparisons/merging happen in the bottom half.

- On each 'level', total number of elements is n.
- And for each merge, $\sharp$comparisons $<$ $\sharp$elements involved.
- So total $\sharp$comparisons for merges on each level is $< n$.
- And there are $\lg n = k$ levels. So total $\sharp$comparisons is $< n \lg n$.

Analysis of MergeSort, ctd.

For simplicity, suppose first that we're sorting a list of size $n = 2^k$. We'll reason informally. Recall our diagram:



All the comparisons/merging happen in the bottom half.

- On each 'level', total number of elements is n.
- And for each merge, $\sharp$comparisons $<$ $\sharp$elements involved.
- So total $\sharp$comparisons for merges on each level is $< n$.
- And there are $\lg n = k$ levels. So total $\sharp$comparisons is $< n \lg n$.

Even if n isn't a power of 2, can show with a little care that $\sharp$comparisons is $< n\lceil \lg n \rceil$, which is certainly $O(n \lg n)$.

MergeSort: worst, best and average case

What about a lower bound?

MergeSort: worst, best and average case

What about a lower bound?

We've seen that on sorted lists of total size m, differing in size by at most 1, **Merge** requires $\Omega(m)$ (actually $\geq m/3$) comparisons.

Using this, can show that on lists of length n, **MergeSort** requires $\Omega(n \lg n)$ comparisons.

MergeSort: worst, best and average case

What about a lower bound?

We've seen that on sorted lists of total size m, differing in size by at most 1, **Merge** requires $\Omega(m)$ (actually $\geq m/3$) comparisons.

Using this, can show that on lists of length n, **MergeSort** requires $\Omega(n \lg n)$ comparisons.

So $T_{worst} = O(n \lg n)$, $T_{best} = \Omega(n \lg n)$.
Can immediately conclude that $T_{worst}, T_{best}, T_{av}$ are all $\Theta(n \lg n)$.
(Shown without deriving exact formulae for $T_{worst}, T_{best}, T_{av}$!)

MergeSort: worst, best and average case

What about a lower bound?

We've seen that on sorted lists of total size m, differing in size by at most 1, **Merge** requires $\Omega(m)$ (actually $\geq m/3$) comparisons.

Using this, can show that on lists of length n, **MergeSort** requires $\Omega(n \lg n)$ comparisons.

So $T_{worst} = O(n \lg n)$, $T_{best} = \Omega(n \lg n)$.
Can immediately conclude that $T_{worst}, T_{best}, T_{av}$ are all $\Theta(n \lg n)$.
(Shown without deriving exact formulae for $T_{worst}, T_{best}, T_{av}$!)

Summary: In worst and average cases, MergeSort is asymptotically better than InsertSort ($n \lg n = o(n^2)$).

But InsertSort does better in best case ($n = o(n \lg n)$).

Measuring ‘overall runtime’

Consider again InsertSort (for integer arrays):

```
InsertSort(A):
    for i = 1 to n−1        # write n for size of A
        x = A[i]
        j = i−1
        while j ≥ 0 and A[j] > x
            A[j+1] = A[j]
            j = j−1
        A[j+1] = x
```

Common to take number of line executions as measure of runtime.

Measuring ‘overall runtime’

Consider again InsertSort (for integer arrays):

```
InsertSort(A):
    for i = 1 to n−1        # write n for size of A
        x = A[i]
        j = i−1
        while j ≥ 0 and A[j] > x
            A[j+1] = A[j]
            j = j−1
        A[j+1] = x
```

Common to take number of line executions as measure of runtime.

Broad justification: Think about how the pseudocode would be implemented on a typical Random Access Machine (RAM).
(Think 32-bit or 64-bit computer . . . except that word size/number of words may be taken as large as required for the given input.)

Measuring ‘overall runtime’

Consider again InsertSort (for integer arrays):

```
InsertSort(A):
    for i = 1 to n−1        # write n for size of A
        x = A[i]
        j = i−1
        while j ≥ 0 and A[j] > x
            A[j+1] = A[j]
            j = j−1
        A[j+1] = x
```

Common to take number of line executions as measure of runtime.

Broad justification: Think about how the pseudocode would be implemented on a typical Random Access Machine (RAM).
(Think 32-bit or 64-bit computer . . . except that word size/number of words may be taken as large as required for the given input.)

Claim: Each line execution takes $\Theta(1)$ time (i.e. between two positive constants $t < t'$). This implies that, for any such impl,

$$\text{total execution time} \;=\; \Theta(\text{number of line executions})$$

Measuring ‘overall runtime’

Consider again InsertSort (for integer arrays):

```
InsertSort(A):
    for i = 1 to n−1      # write n for size of A
        x = A[i]
        j = i−1
        while j ≥ 0 and A[j] > x
            A[j+1] = A[j]
            j = j−1
        A[j+1] = x
```

Common to take number of line executions as measure of runtime.

Broad justification: Think about how the pseudocode would be implemented on a typical Random Access Machine (RAM).
(Think 32-bit or 64-bit computer ... except that word size/number of words may be taken as large as required for the given input.)

Claim: Each line execution takes $\Theta(1)$ time (i.e. between two positive constants $t < t'$). This implies that, for any such impl,

$$\text{total execution time} \;=\; \Theta(\text{number of line executions})$$

Warning! Applies only when each line execution does just a bounded amount of work. E.g. What if ‘$>$’ is comparison for strings?

Overall runtime of InsertSort and MergeSort

Can do ‘line execution’ analyses of **InsertSort** and **MergeSort** using same ideas as before.

Overall runtime of InsertSort and MergeSort

Can do 'line execution' analyses of **InsertSort** and **MergeSort** using same ideas as before.

E.g. In **InsertSort**, for each value of i, execution of lines 2–7 takes $\leq 3i + 4$ line executions . . .

Overall runtime of InsertSort and MergeSort

Can do 'line execution' analyses of **InsertSort** and **MergeSort** using same ideas as before.

E.g. In **InsertSort**, for each value of i, execution of lines 2–7 takes $\leq 3i + 4$ line executions ...

In this case, this tells the same story as our previous analyses:

	Worst	Average	Best
InsertSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
MergeSort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$

Space complexity

Let's also look briefly at the memory requirements of our algorithms on size n inputs.

Sensible to ignore the space occupied by the input array A, and consider the extra space needed.

Space complexity

Let's also look briefly at the memory requirements of our algorithms on size n inputs.

Sensible to ignore the space occupied by the input array A, and consider the extra space needed.

Easy to see that . . .

- 'External' InsertSort (putting result in new array B) requires $\Theta(n)$ extra space.

Space complexity

Let's also look briefly at the memory requirements of our algorithms on size n inputs.

Sensible to ignore the space occupied by the input array A, and consider the extra space needed.

Easy to see that ...

- 'External' InsertSort (putting result in new array B) requires $\Theta(n)$ extra space.
- In-place InsertSort requires only $\Theta(1)$ extra space (just i,j,x).

Space complexity

Let's also look briefly at the memory requirements of our algorithms on size n inputs.

Sensible to ignore the space occupied by the input array A, and consider the extra space needed.

Easy to see that ...

- 'External' InsertSort (putting result in new array B) requires $\Theta(n)$ extra space.
- In-place InsertSort requires only $\Theta(1)$ extra space (just i,j,x).
- MergeSort apparently requires $\Theta(n \log n)$ space (total size of arrays created).

Space complexity

Let's also look briefly at the memory requirements of our algorithms on size n inputs.

Sensible to ignore the space occupied by the input array A, and consider the extra space needed.

Easy to see that ...

- 'External' InsertSort (putting result in new array B) requires $\Theta(n)$ extra space.
- In-place InsertSort requires only $\Theta(1)$ extra space (just i,j,x).
- MergeSort apparently requires $\Theta(n \log n)$ space (total size of arrays created).
- However, with more careful memory management, MergeSort can be implemented using just $\Theta(n)$ extra space.
  Idea is that after doing D = **Merge** (B,C), space occupied by 'temporary' arrays B,C can be reclaimed.

Space complexity

Let's also look briefly at the memory requirements of our algorithms on size n inputs.

Sensible to ignore the space occupied by the input array A, and consider the extra space needed.

Easy to see that ...

- 'External' InsertSort (putting result in new array B) requires $\Theta(n)$ extra space.
- In-place InsertSort requires only $\Theta(1)$ extra space (just i,j,x).
- MergeSort apparently requires $\Theta(n \log n)$ space (total size of arrays created).
- However, with more careful memory management, MergeSort can be implemented using just $\Theta(n)$ extra space.
  Idea is that after doing D = **Merge** (B,C), space occupied by 'temporary' arrays B,C can be reclaimed.

Headline: In-place InsertSort wins on space efficiency.

'End of Part 1'

That completes our introduction to the general concepts that underpin this course, and the way we'll be approaching algorithms.

'End of Part 1'

That completes our introduction to the general concepts that underpin this course, and the way we'll be approaching algorithms.

A few reflections on the role of asymptotics...

- Offers *one* helpful perspective on algorithms (among others).

‘End of Part 1’

That completes our introduction to the general concepts that underpin this course, and the way we’ll be approaching algorithms.

A few reflections on the role of asymptotics...

- Offers *one* helpful perspective on algorithms (among others).
- Gives us the main headlines, e.g. ‘$T_{Mergesort}(n) = \Theta(n \lg n)$’. This says something robust and portable, and can be shown without too much detailed work.

'End of Part 1'

That completes our introduction to the general concepts that underpin this course, and the way we'll be approaching algorithms.

A few reflections on the role of asymptotics...

- Offers *one* helpful perspective on algorithms (among others).
- Gives us the main headlines, e.g. '$T_{Mergesort}(n) = \Theta(n \lg n)$'. This says something robust and portable, and can be shown without too much detailed work.
- But suppresses detail which may matter in practice. (May sometimes say e.g. '$T(n) = O(n \lg n)$ but with large hidden constants').

'End of Part 1'

That completes our introduction to the general concepts that underpin this course, and the way we'll be approaching algorithms.

A few reflections on the role of asymptotics...

- Offers *one* helpful perspective on algorithms (among others).
- Gives us the main headlines, e.g. '$T_{Mergesort}(n) = \Theta(n \lg n)$'. This says something robust and portable, and can be shown without too much detailed work.
- But suppresses detail which may matter in practice. (May sometimes say e.g. '$T(n) = O(n \lg n)$ but with large hidden constants').
- Usually reliable: if asymptotics predicts 'A is better than B', this can usually be observed on realistic inputs. (There are exceptions: 'galactic algorithms'.)

‘End of Part 1’

That completes our introduction to the general concepts that underpin this course, and the way we’ll be approaching algorithms.

A few reflections on the role of asymptotics...

- Offers *one* helpful perspective on algorithms (among others).
- Gives us the main headlines, e.g. ‘$T_{Mergesort}(n) = \Theta(n \lg n)$’. This says something robust and portable, and can be shown without too much detailed work.
- But suppresses detail which may matter in practice. (May sometimes say e.g. ‘$T(n) = O(n \lg n)$ but with large hidden constants’).
- Usually reliable: if asymptotics predicts ‘A is better than B’, this can usually be observed on realistic inputs. (There are exceptions: ‘galactic algorithms’.)

Next lecture: Start on data structures — beginning with how program data is organized in memory.

Reading

Roughgarden 1.5, 1.6
Kleinberg-Tardos 2.4 (first half), 4.1
CLRS 2.2