2301520 FUNDAMENTALS OF AMCS

Lecture 5: Sorting

Lectured by Dr. Krung Sinapiromsaran,
Krung.S@chula.ac.th

Excerpted from Dr. William Smith,
wsmith@cs.york.ac.uk

Outline

- What is sorting and why is it important?
- Simple quadratic sorts (selection and insertion sort)
- An O(n log n) sort using divide-and-conquer (merge sort)
- Limits of sorting – comparison-based algorithms
- Doing even better – non-comparison-based

Objective

- Be able to explain the usage of sorting in various problem
- Track the algorithm performance
- Analyze various sorting algorithms
- Compare sorting algorithms with respect to asymptotic notation
- Explain the limitation of comparison-based sorting algorithms
- Explain why some sorting algorithms can achieve a better running time

The problem of sorting

- Input:A sequence $s = \langle e_1, ..., e_n \rangle$ of n elements

The problem of sorting

- Input:A sequence $s = \langle e_1, ..., e_n \rangle$ of n elements

We make the following assumptions:

- Each element has an associated key: $k_i = \text{key}(e_i)$
- There is a linear order defined on the keys: $\leq$
- For ease of notation we write: $e \leq e' \leftrightarrow \text{key}(e) \leq \text{key}(e')$

The problem of sorting

- Input: A sequence $s = \langle e_1, ..., e_n \rangle$ of n elements

We make the following assumptions:

  - Each element has an associated key: $k_i = \text{key}(e_i)$
  - There is a linear order defined on the keys: $\leq$
  - For ease of notation we write: $e \leq e' \leftrightarrow \text{key}(e) \leq \text{key}(e')$

- Output: A sequence $s' = \langle e'_1, ..., e'_n \rangle$

  where s' is a permutation of s and $e'_1 \leq ... \leq e'_n$

The problem of sorting

- Input:A sequence $s = \langle e_1, ..., e_n \rangle$ of n elements

We make the following assumptions:

  - Each element has an associated key: $k_i = \text{key}(e_i)$
  - There is a linear order defined on the keys: $\leq$
  - For ease of notation we write: $e \leq e' \leftrightarrow \text{key}(e) \leq \text{key}(e')$

- Output:A sequence $s' = \langle e'_1, ..., e'_n \rangle$

  where s' is a permutation of s and $e'_1 \leq ... \leq e'_n$

- Sorting as a formal postcondition:

  For all $i, j, 0 < i < j \leq n \Rightarrow s'[i] < s'[j]$ & $\text{bag}(s) = \text{bag}(s')$

Aside:Bags or Multisets

- Two informal definitions of a bag or multiset:
  - A bag is like a set, but a member of a bag can have more than one membership
  - A bag is like a sequence with the order thrown away
- Intuitive analogy: think of coins in your pocket

- An element is either in a set or it isn't
- An element belongs to a bag zero or more times
- The number of times an element belongs to a bag is its ***multiplicity***
- The total number of elements in a bag, including repeated memberships, is its cardinality

Motivation

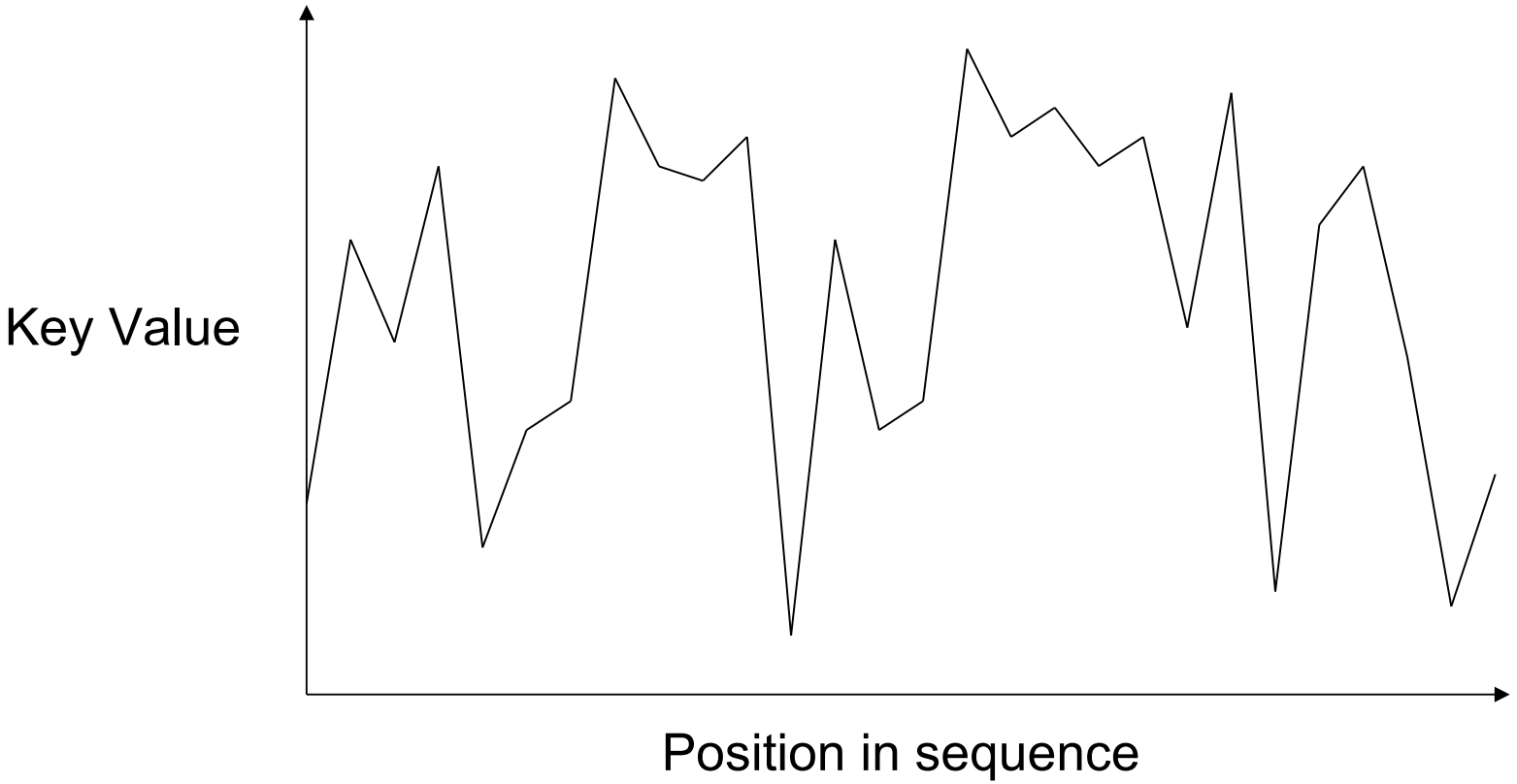
- Sorting is standard fodder for an introduction to algorithms course
- Good for teaching algorithm analysis, lots of alternative algorithms to compare, different algorithmic concepts (divide-and-conquer, randomised algorithms etc)
- This is all good, but beyond this, is sorting very important in the real world?
- Answer: yes, definitely!
- According to Skiena, computers spend approx. 25% of their time sorting, so doing this efficiently is hugely important
- New developments are still being made in this area (library sort in 2004)
- There are a huge number of applications of sorting – it makes lots of other tasks possible or easier or more efficient

Applications of Sorting

- **Search preprocessing**: we saw last time that sorting an array reduces searching from O(n) to O(log n)
- **Selection**: what is the k^{th} largest element in a sequence? If it's sorted, just pick element k.
- **Convex hulls**: what is the smallest polygon that encloses a set of points? (useful in geometric algorithms – computer graphics and vision). To solve, sort points by x-coordinate, add from left to right, delete points when enclosed by polygon including new point.
- **Closest pair**: given a set of numbers, find the pair with the smallest difference between them. To solve: sort, then just do a linear scan through the sequence, keeping track of smallest distance so far.
- Interesting note: IBM was formed principally on being able to sort US census data

Hedgehog Diagram (1)

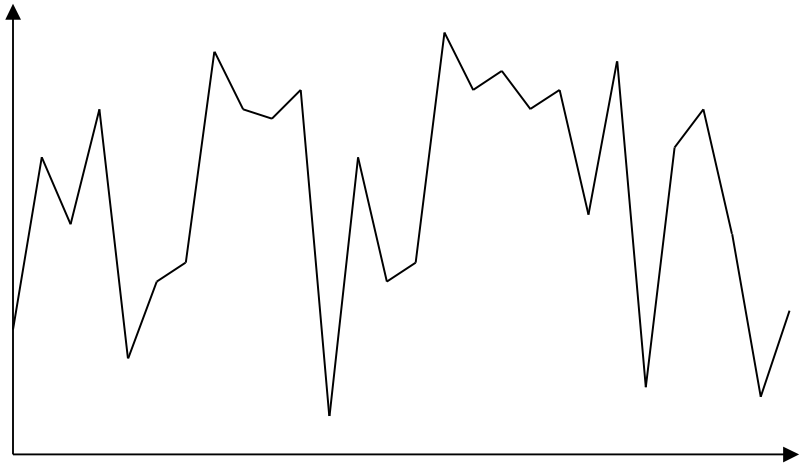
- A useful visualization of our data:



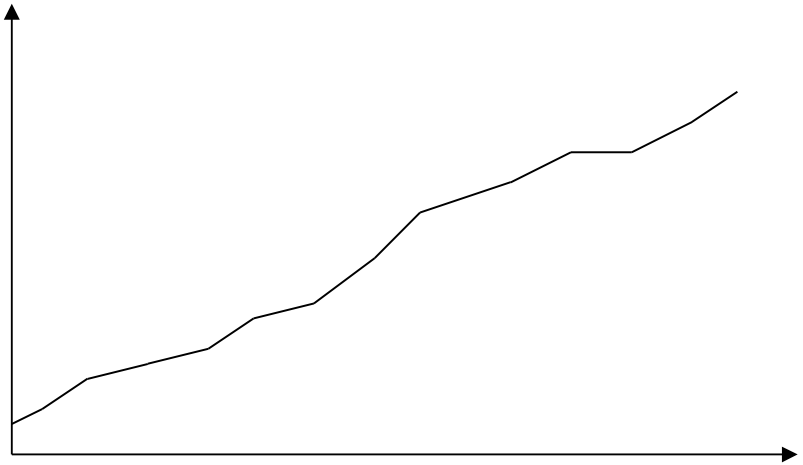
Hedgehog Diagram (2)

- We can visualise the precondition and postcondition of the sorting problem in terms of these diagrams:

Precondition



Postcondition



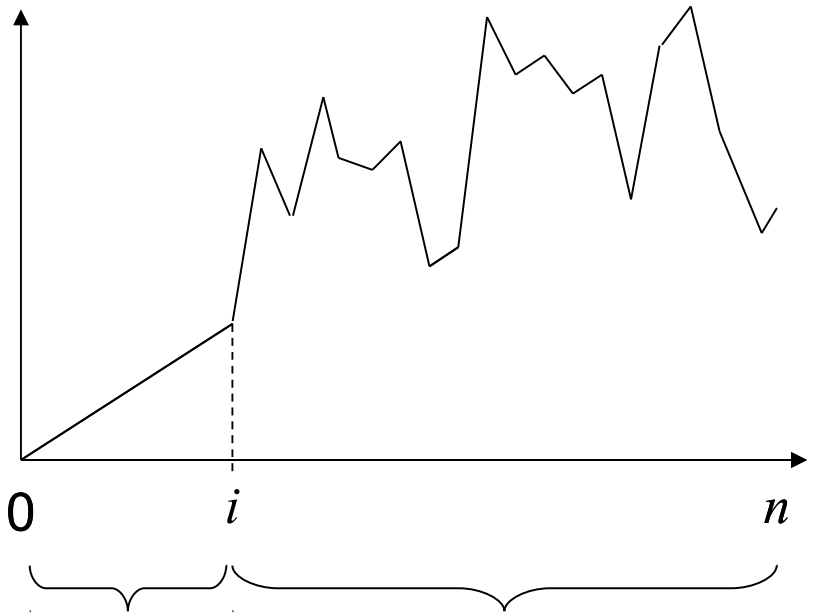
- It can also be helpful to visualise the intermediate states of sorting algorithms using these diagrams – visualise the invariant preserved

Selection sort concept

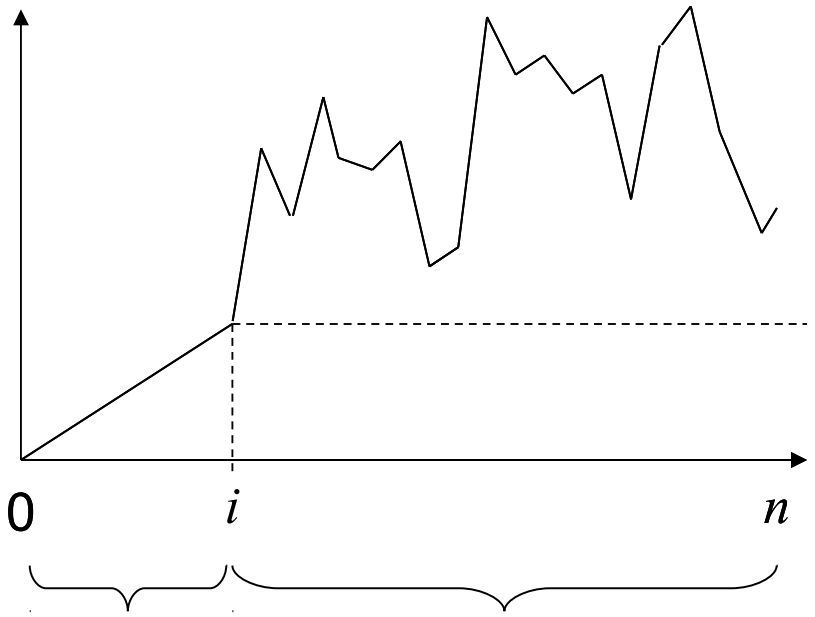
- Our first sorting algorithm is selection sort.
- The idea is to maintain the following invariant:
  - The partially sorted sequence consists of two parts:
    - The first part, which is already sorted.
    - The second part, which is unsorted.
  - Moreover, all elements in the second part are larger than all those in the first part

Selection sort graph (1)

Selection sort graph (2)

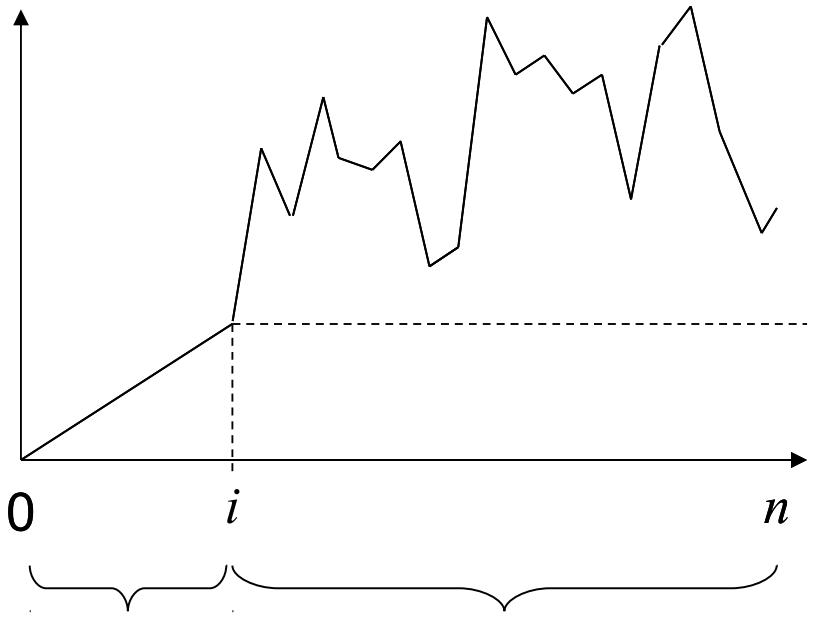


Selection sort graph (3)



All elements still to be sorted are greater than the largest sorted element

Selection sort graph (4)



- Selection sort works by repeatedly selecting the smallest element from the unsorted part and adding it to the top of the sorted part

Selection sort algorithm (1)

```
Function SelectionSort(A[1:n]:array of elements)
1. for i := 1 to n – 1 do
2.       Invariant A[1] ≤ … ≤ A[i]
3.       min := i
4.       for j := i+1 to n do
5.           if A[j] < A[min] then
6.               min := j
7.           endif
8.       endfor
9.       swap(A[i], A[min])
10.      invariant max(A[1..i]) ≤ A[i+1..n]
11. endfor
```

Selection sort algorithm (2)

```
Function SelectionSort(A[1:n]:array of elements)
1. for i := 1 to n – 1 do
2.       Invariant A[1] ≤ … ≤ A[i]
3.       min := i
4.       for j := i+1 to n do
5.           if A[j] < A[min] then
6.               min := j
7.           endif
8.       endfor
9.       swap(A[i], A[min])
10.      invariant max(A[1..i]) ≤ A[i+1..n]
11. endfor
```

Min stores the index of the Minimum element found so Far in A[i+1..n]

Selection sort algorithm (3)

```
Function SelectionSort(A[1:n]:array of elements)
1. for i := 1 to n – 1 do
2.       Invariant A[1] ≤ … ≤ A[i]
3.       min := i
4.       for j := i+1 to n do
5.           if A[j] < A[min] then
6.               min := j
7.           endif
8.       endfor
9.       swap(A[i], A[min])
10.      invariant max(A[1..i]) ≤ A[i+1..n]
11. endfor
```

Min stores the index of the Minimum element found so Far in A[i+1..n]

This loop finds the Minimum element in A[i+1..n]

Selection sort algorithm (4)

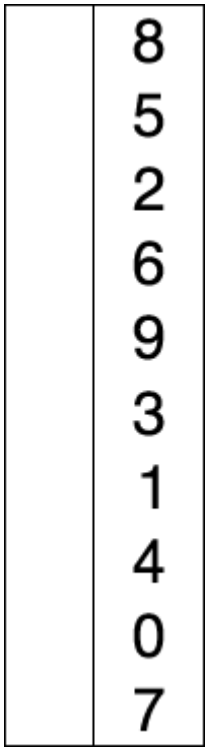
```
Function SelectionSort(A[1:n]:array of elements)
1. for i := 1 to n – 1 do
2.       Invariant A[1] ≤ … ≤ A[i]
3.       min := i
4.       for j := i+1 to n do
5.           if A[j] < A[min] then
6.               min := j
7.           endif
8.       endfor
9.       swap(A[i], A[min])
10.      invariant max(A[1..i]) ≤ A[i+1..n]
11. endfor
```

Min stores the index of the Minimum element found so Far in A[i+1..n]

This loop finds the Minimum element in A[i+1..n]

This puts the newly found minimum into position i restoring the invariant

Selection sort animation



Asymptotic analysis of the selection sort

- What is the complexity of selection sort?
- The main body consists of two nested loops
- We know how to analyze nested loops

Selection sort analysis (1)

```
Function SelectionSort(A[1:n]:array of elements)
1. for i := 1 to n – 1 do
2.       Invariant A[1] ≤ … ≤ A[i]
3.       min := i
4.       for j := i+1 to n do
5.           if A[j] < A[min] then
6.               min := j
7.           endif
8.       endfor
9.       swap(A[i], A[min])
10.      invariant max(A[1..i]) ≤ A[i+1..n]
11. endfor
```

Inside the inner loop we do O(1) of work

Total work for loop: $\sum_{j=i+1}^{n} 1 = n-(i+1)$

Selection sort analysis (2)

```
Function SelectionSort(A[1:n]:array of elements)
1. for i := 1 to n – 1 do
2.       Invariant A[1] ≤ … ≤ A[i]
3.       min := i
4.       for j := i+1 to n do
5.           if A[j] < A[min] then
6.               min := j
7.           endif
8.       endfor
9.       swap(A[i], A[min])
10.      invariant max(A[1..i]) ≤ A[i+1..n]
11. endfor
```

Inside the outer loop we do O(1) work plus the inner loop:

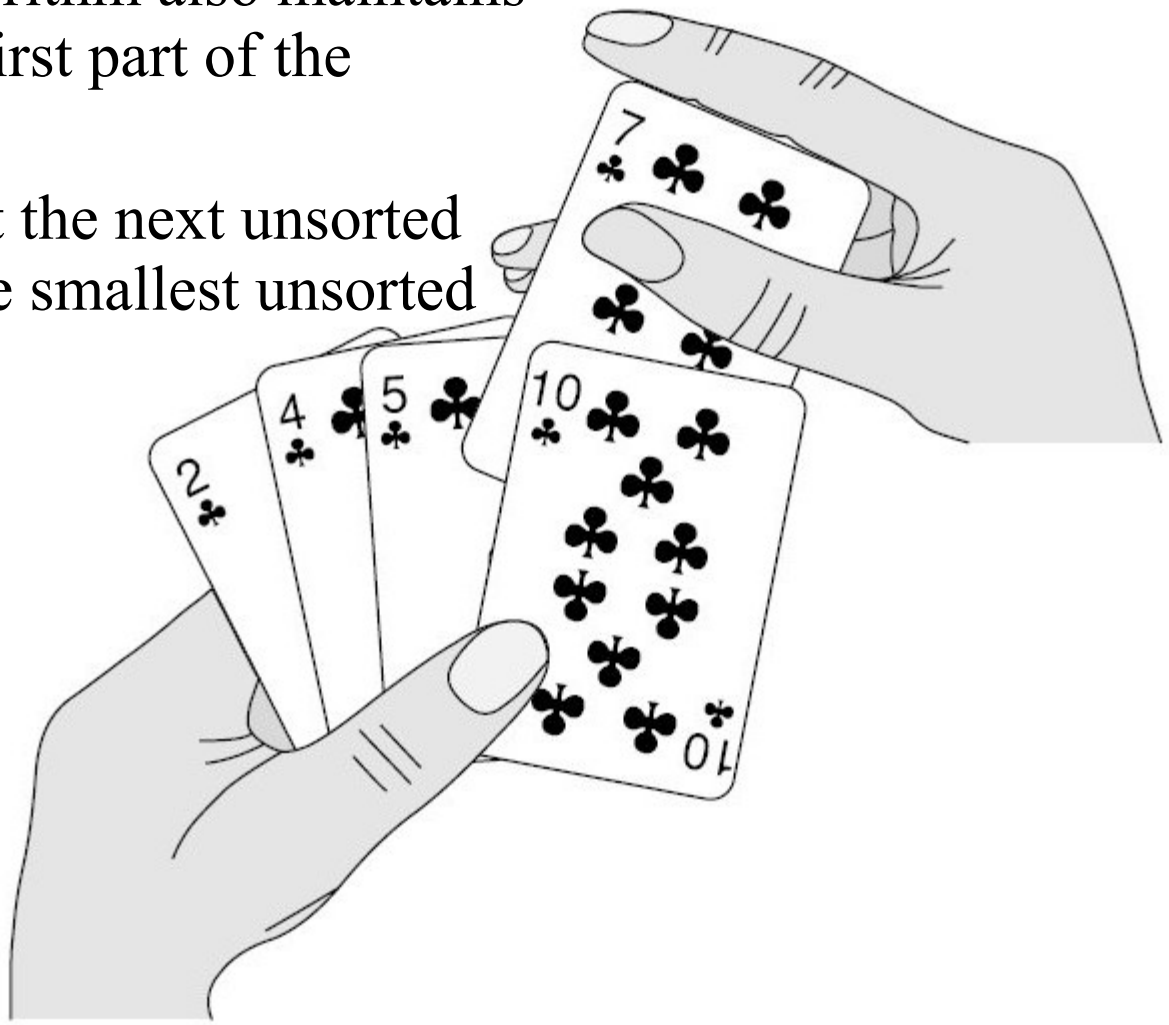
$$\sum_{i=1}^{n-1} n-(i+1)=\Theta(n^2)$$

O(n^2) Selection sort

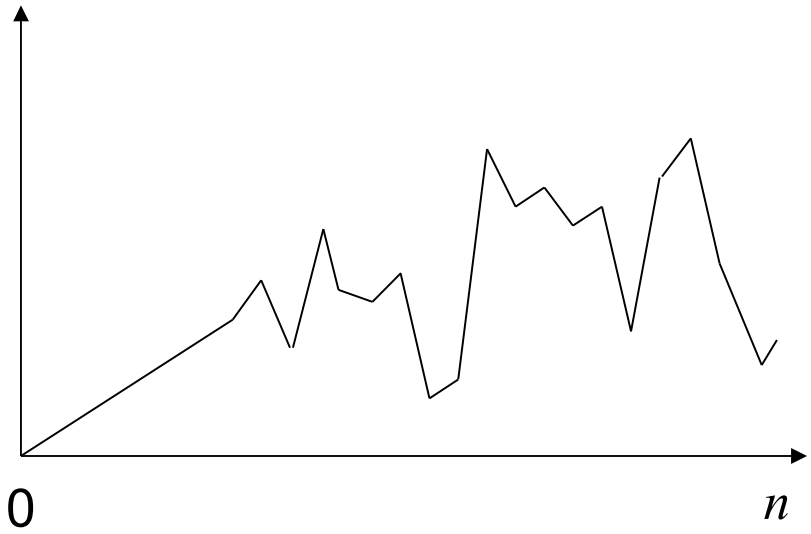
- So selection sort is an O(n^2) algorithm
- We know such algorithms are useable for up to about a million items
- If we relax the invariant slightly, we obtain another O(n^2) algorithm

Insertion sort concept

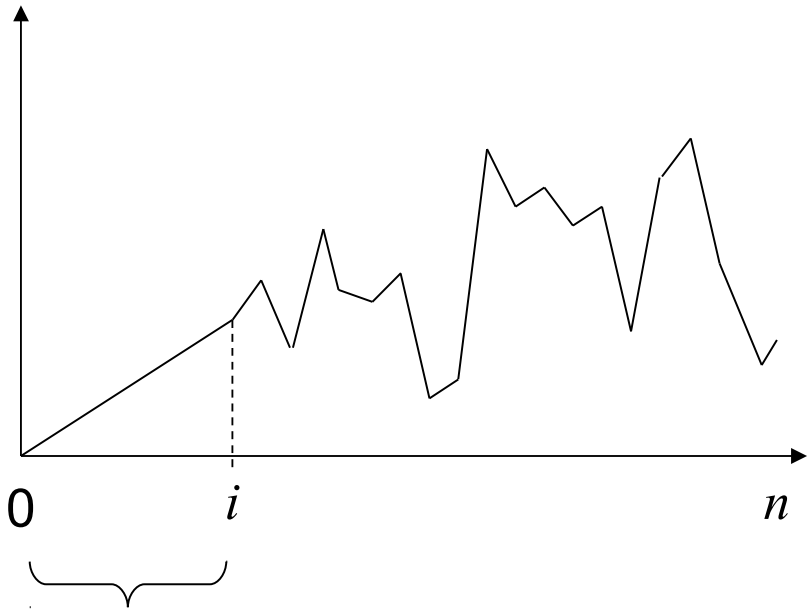
- The insertion sort algorithm also maintains the invariant that the first part of the sequence is sorted.
- But this time we insert the next unsorted element rather than the smallest unsorted element



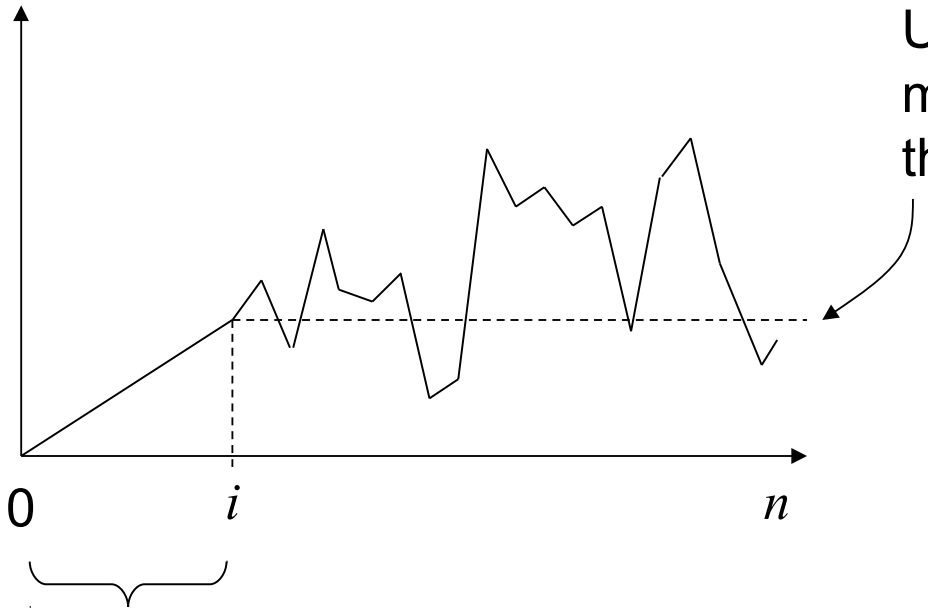
Insertion sort graph (1)



Insertion sort graph (2)



Insertion sort graph (3)



Insertion sort algorithm (1)

```
Function InserionSort(A[1:n]:array of elements)
1. for i := 2 to n do
2.       Invariant A[1] ≤ … ≤ A[i-1]
3.       value := A[i]
4.       j := i - 1
5.       while j > 0 and A[j] > value do
6.                A[j + 1] := A[j]
7.                j := j – 1
8.       endwhile
9.       A[j+1] = value
10. endfor
```

Insertion sort algorithm (2)

```
Function InserionSort(A[1:n]:array of elements)
1. for i := 2 to n do
2.       Invariant A[1] ≤ … ≤ A[i-1]
3.       value := A[i]
4.       j := i - 1
5.       while j > 0 and A[j] > value do
6.             A[j + 1] := A[j]
7.             j := j – 1
8.       endwhile
9.       A[j+1] = value
10. endfor
```

value contains the value of the element to insert

Insertion sort algorithm (3)

```
Function InserionSort(A[1:n]:array of elements)
1. for i := 2 to n do
2.       Invariant A[1] ≤ … ≤ A[i-1]
3.       value := A[i]
4.       j := i - 1
5.       while j > 0 and A[j] > value do
6.             A[j + 1] := A[j]
7.             j := j – 1
8.       endwhile
9.       A[j+1] = value
10. endfor
```

value contains the value of the element to insert

This loops down through the sorted part of the sequence, shuffling elements up and stopping when we find an element less than the element to insert

Insertion sort algorithm (4)

```
Function InserionSort(A[1:n]:array of elements)
1. for i := 2 to n do
2.       Invariant A[1] ≤ … ≤ A[i-1]
3.       value := A[i]
4.       j := i - 1
5.       while j > 0 and A[j] > value do
6.             A[j + 1] := A[j]
7.             j := j – 1
8.       endwhile
9.       A[j+1] = value
10. endfor
```

value contains the value of the element to insert

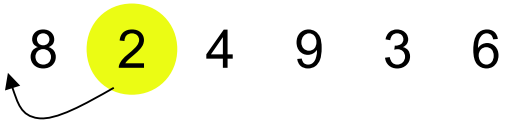
This loops down through the sorted part of the sequence, shuffling elements up and stopping when we find an element less than the element to insert

We can then insert the element restoring the invariant

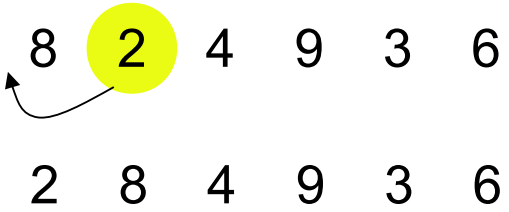
Insertion sort example (1)

8 2 4 9 3 6

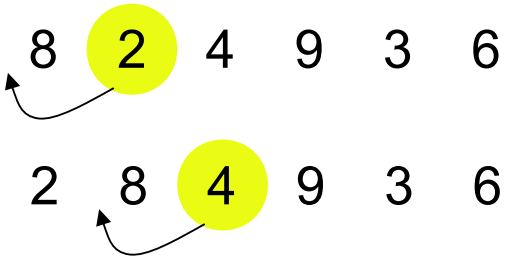
Insertion sort example (2)



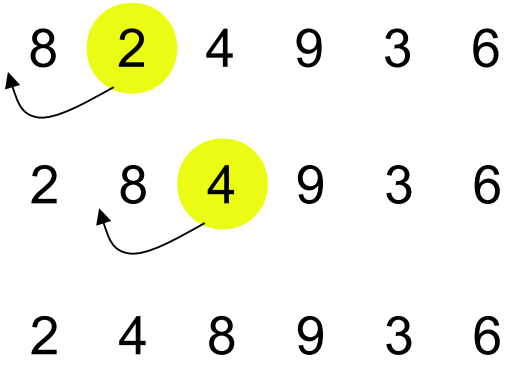
Insertion sort example (3)



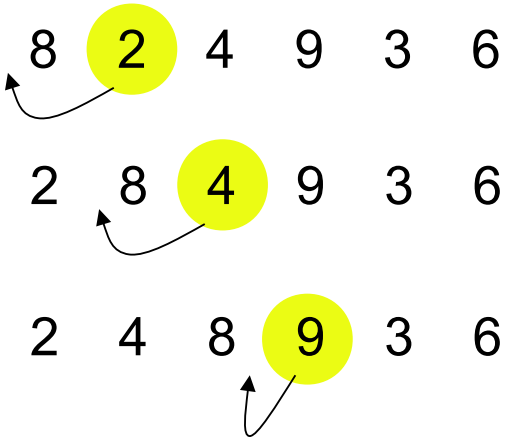
Insertion sort example (4)



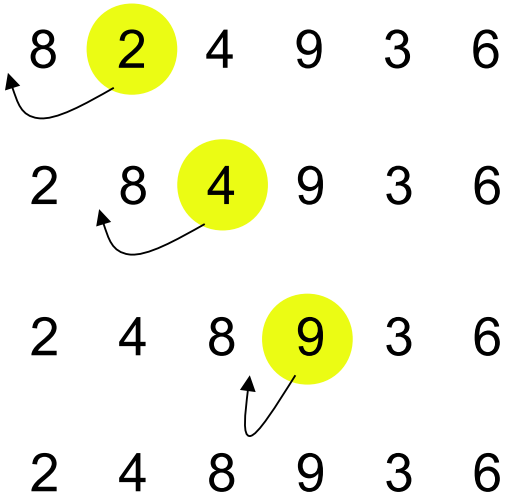
Insertion sort example (5)



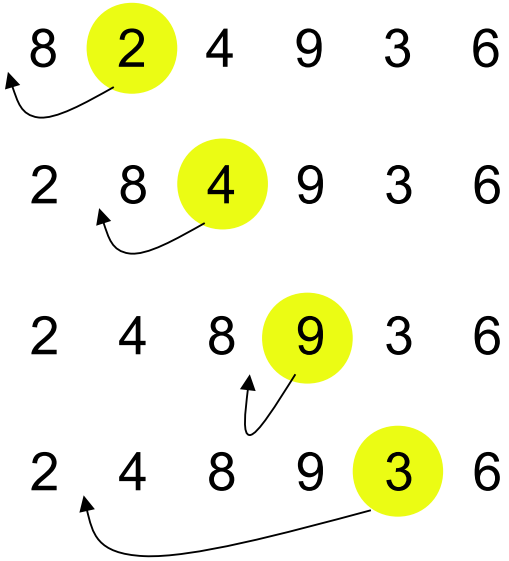
Insertion sort example (6)



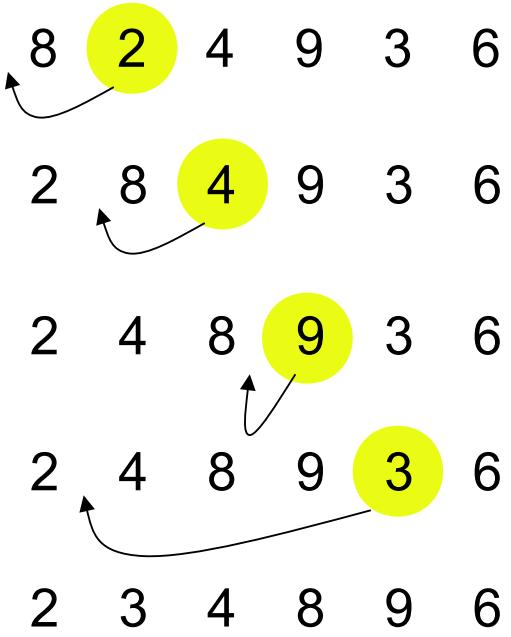
Insertion sort example (7)



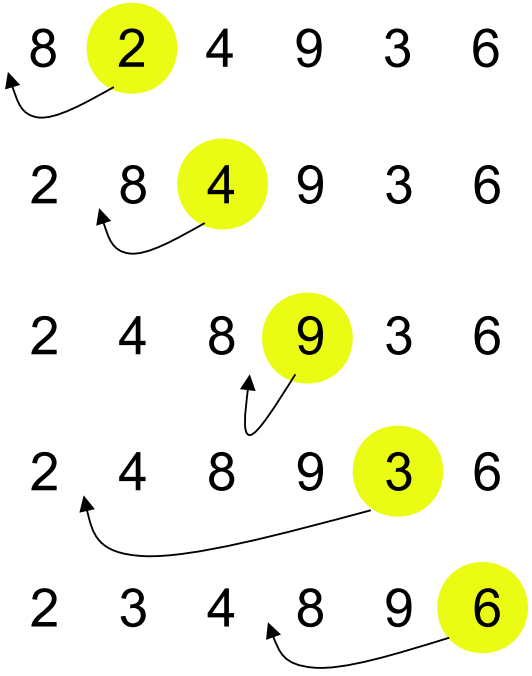
Insertion sort example (8)



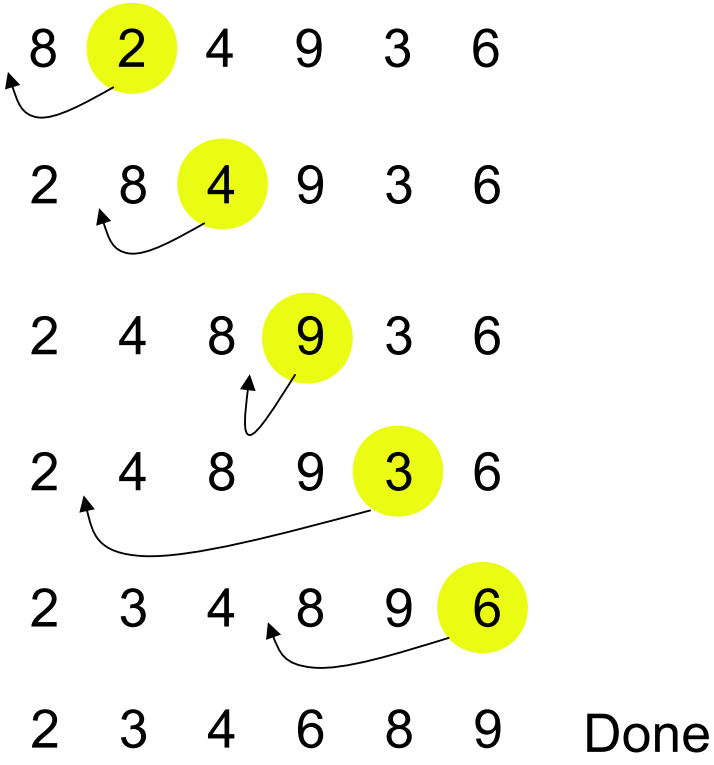
Insertion sort example (9)



Insertion sort example (10)



Insertion sort example (11)



Insertion sort algorithm (5)

```
Function InserionSort(A[1:n]:array of elements)
1. for i := 2 to n do
2.       Invariant A[1] ≤ … ≤ A[i-1]
3.       value := A[i]
4.       j := i - 1
5.       while j > 0 and A[j] > value do
6.             A[j + 1] := A[j]
7.             j := j – 1
8.       endwhile
9.       A[j+1] = value
10. endfor
```

We don't know in advance how many times this loop will iterate

Q:In the worst case?

Q:What is happening in this case?

Insertion sort algorithm (6)

```
Function InserionSort(A[1:n]:array of elements)
1. for i := 2 to n do
2.       Invariant A[1] ≤ … ≤ A[i-1]
3.       value := A[i]
4.       j := i - 1
5.       while j > 0 and A[j] > value do
6.              A[j + 1] := A[j]
7.              j := j – 1
8.       endwhile
9.       A[j+1] = value
10. endfor
```

We don't know in advance how many times this loop will iterate

Q:In the worst case?

Q:What is happening in this case?

A:The worst case occurs when the element to be inserted is smaller than all the elements sorted so far

In this case the loop will executed i-1 times: $\sum_{j=0}^{i-1} 1 = i - 1$

Insertion sort algorithm (7)

```
Function InserionSort(A[1:n]:array of elements)
1. for i := 2 to n do
2.       Invariant A[1] ≤ … ≤ A[i-1]
3.       value := A[i]
4.       j := i - 1
5.       while j > 0 and A[j] > value do
6.              A[j + 1] := A[j]
7.              j := j – 1
8.       endwhile
9.       A[j+1] = value
10. endfor
```

For the total complexity we then substitute the worst case for the inner loop into the outer loop:

$$\sum_{i=2}^{n} i-1=\Theta\left(n^2\right)$$

So in the worst case, insertion sort is also an n^2 algorithm

Q:When does the worst case occur?

Q:When does the best case occur? What is the complexity in that case?

Insertion sort worst case

- Insertion sort's worst case occurs when given a reverse sorted sequence
- In this case, each insertion requires shuffling all the way down the sorted part of the sequence
- Insertion sort's best case occurs when given a sorted sequence
- In this case it is linear
- It is also good on a partially sorted sequence
- In practice it is more efficient than other simple quadratic sorters (Selection sort)
- For small datasets, because of a small leading constant on the running time, it is a good choice of sorting algorithm
- Nevertheless, we can do better

Divide-and-conquer sorting (1)

- Q:Try to think about sorting in terms of divide-and-conquer. Express sorting as a divide-and-conquer algorithm in the way that seems most natural to you.

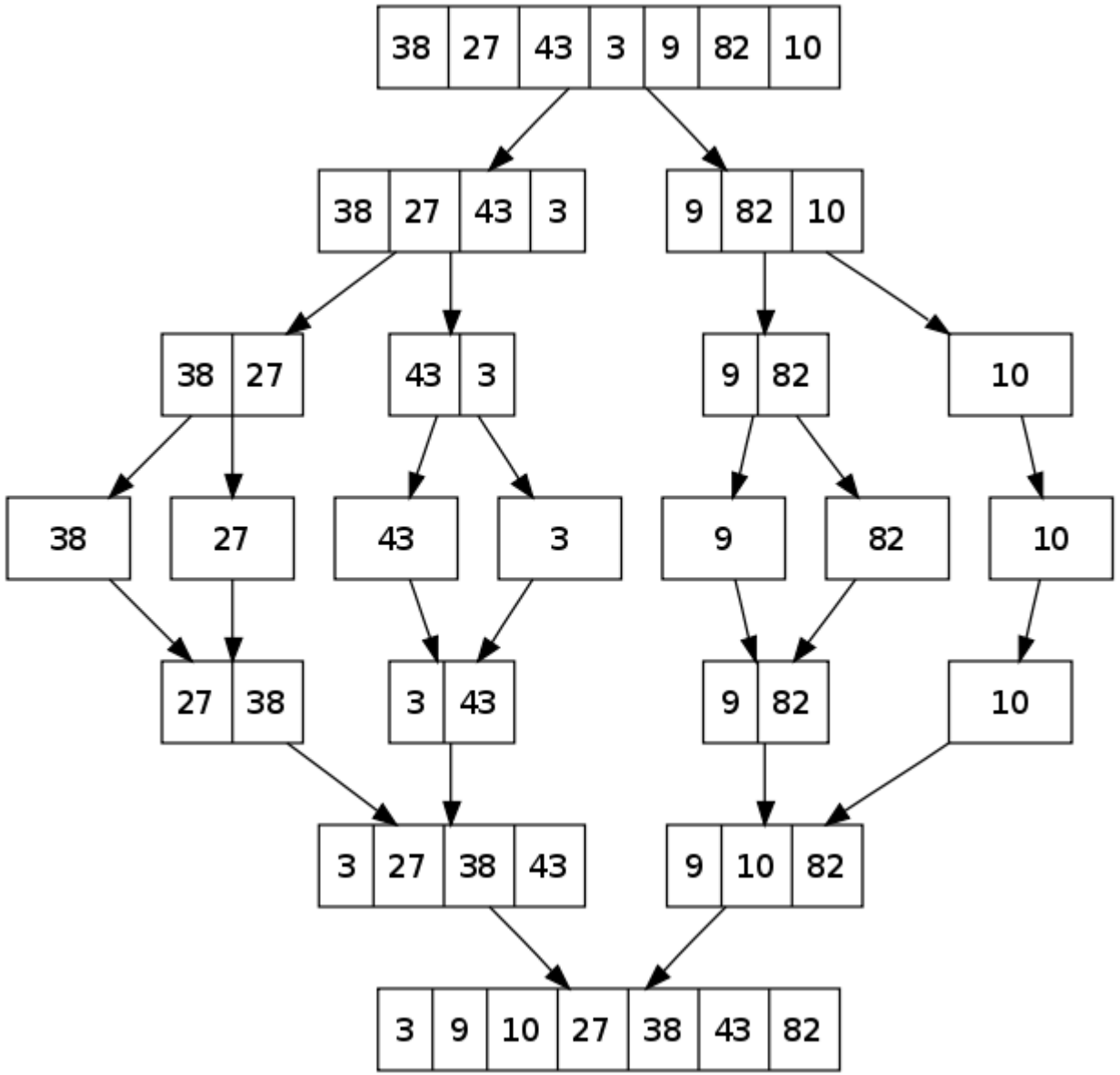
Divide-and-conquer sorting (2)

- Q:Try to think about sorting in terms of divide-and-conquer. Express sorting as a divide-and-conquer algorithm in the way that seems most natural to you.
- A:Here's one answer which seems natural:
- Split the input sequence into two halves. Recursively sort each of those halves and then merge the two sorted halves into one.
- This is merge sort.
- An elegant idea which is easy to express, but is it efficient?

Merge sort algorithm

```
Function MergeSort(A[1:n]:array of elements)
1. if n = 1 then
2.    return A
3. else
4.    A1 = MergeSort(A[1..n/2])
5.    A2 = MergeSort(A[n/2..n])
6.    return merge(A1, A2)
7. endif
```

Merge sort Example



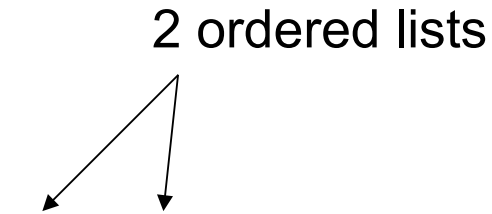
Merge sort analysis

```
Function MergeSort(A[1:n]:array of elements)
1. if n = 1 then
2.    return A
3. else
4.    A1 = MergeSort(A[1..n/2])
5.    A2 = MergeSort(A[n/2..n])
6.    return merge(A1, A2)
7. endif
```

To calculate the complexity of merge sort, we need to answer two questions:

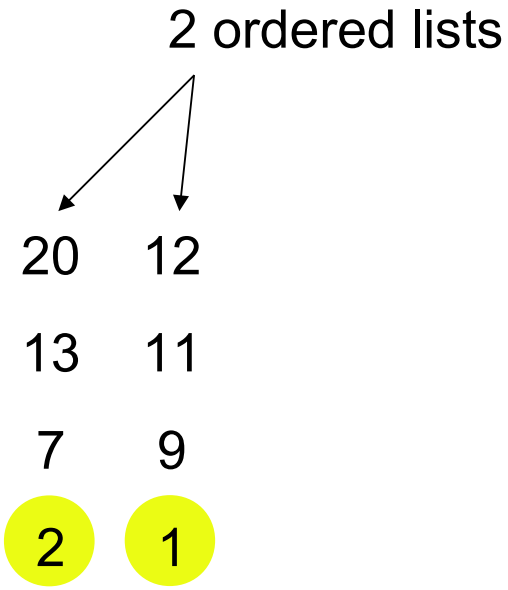
1. What is the complexity of the merge subroutine?
2. What is the solution to the recurrence relation resulting from the recursive call?

Merge 2 ordered lists (1)



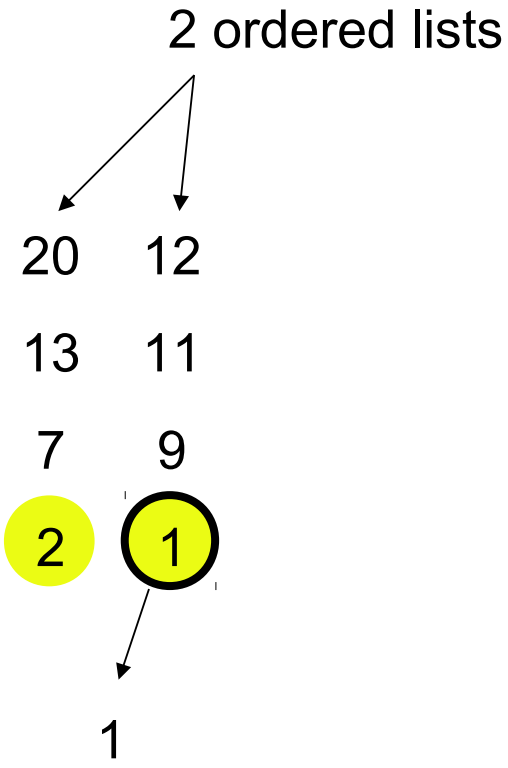
20	12
13	11
7	9
2	1

Merge 2 ordered lists (2)



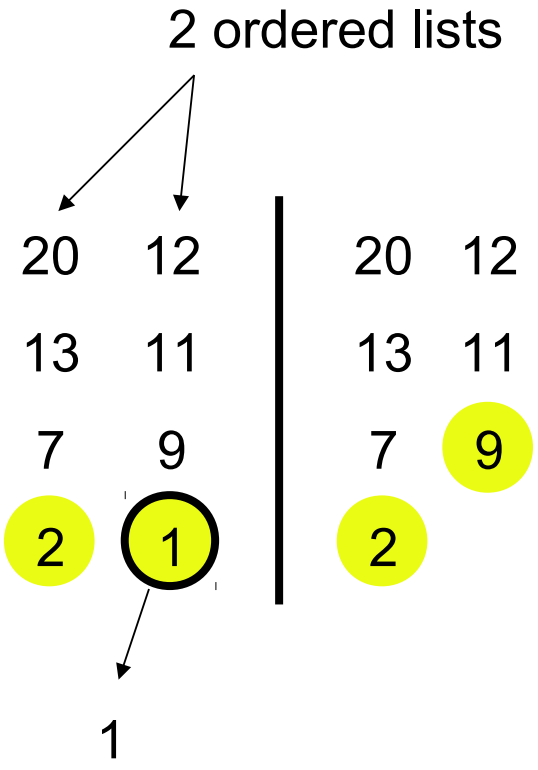
Strategy: examine front of both lists, repeatedly remove smallest

Merge 2 ordered lists (3)



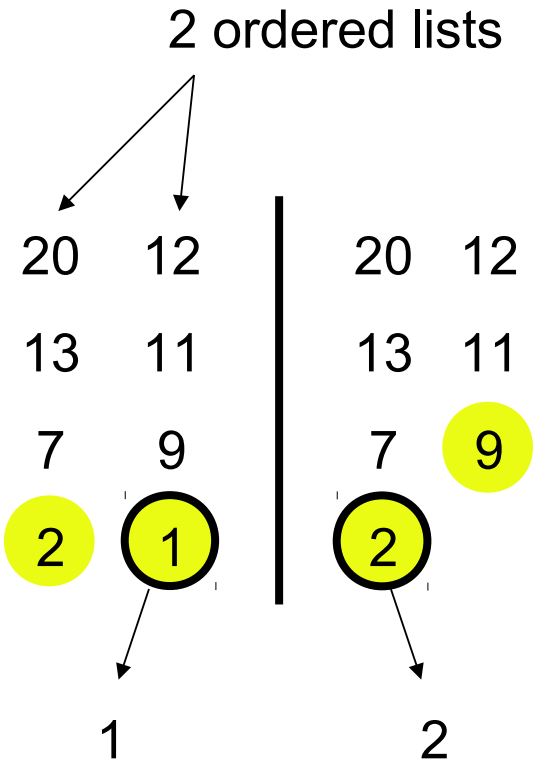
Strategy: examine front of both lists, repeatedly remove smallest

Merge 2 ordered lists (4)



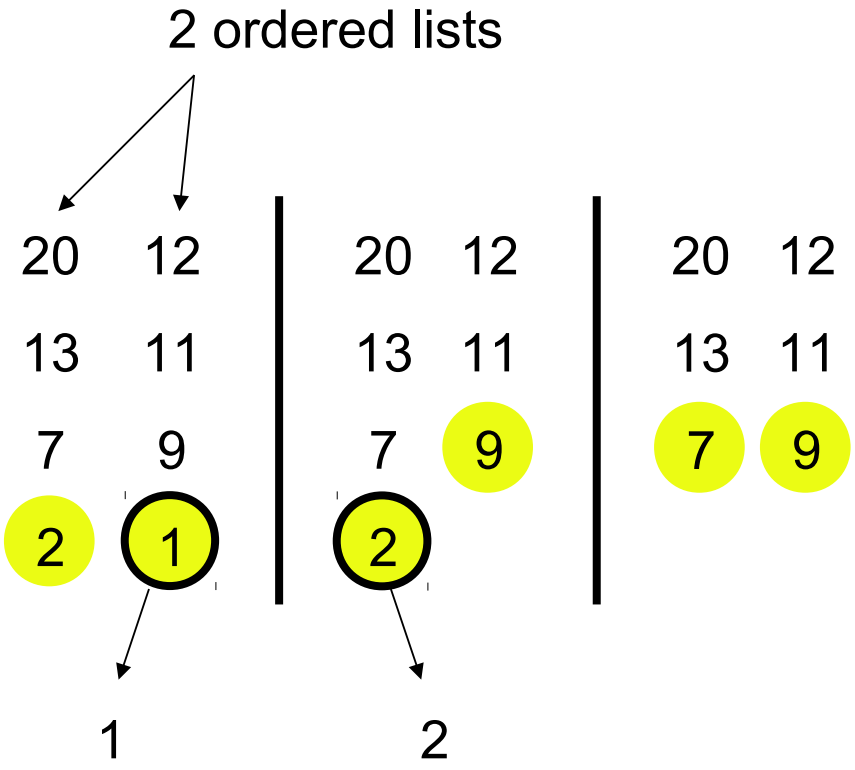
Strategy: examine front of both lists, repeatedly remove smallest

Merge 2 ordered lists (5)



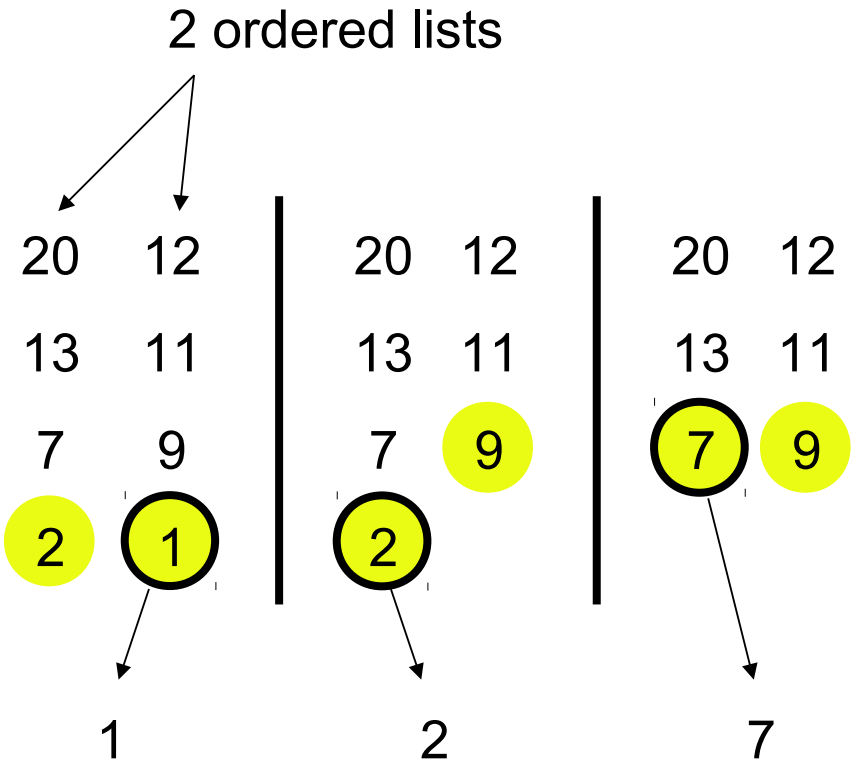
Strategy: examine front of both lists, repeatedly remove smallest

Merge 2 ordered lists (6)



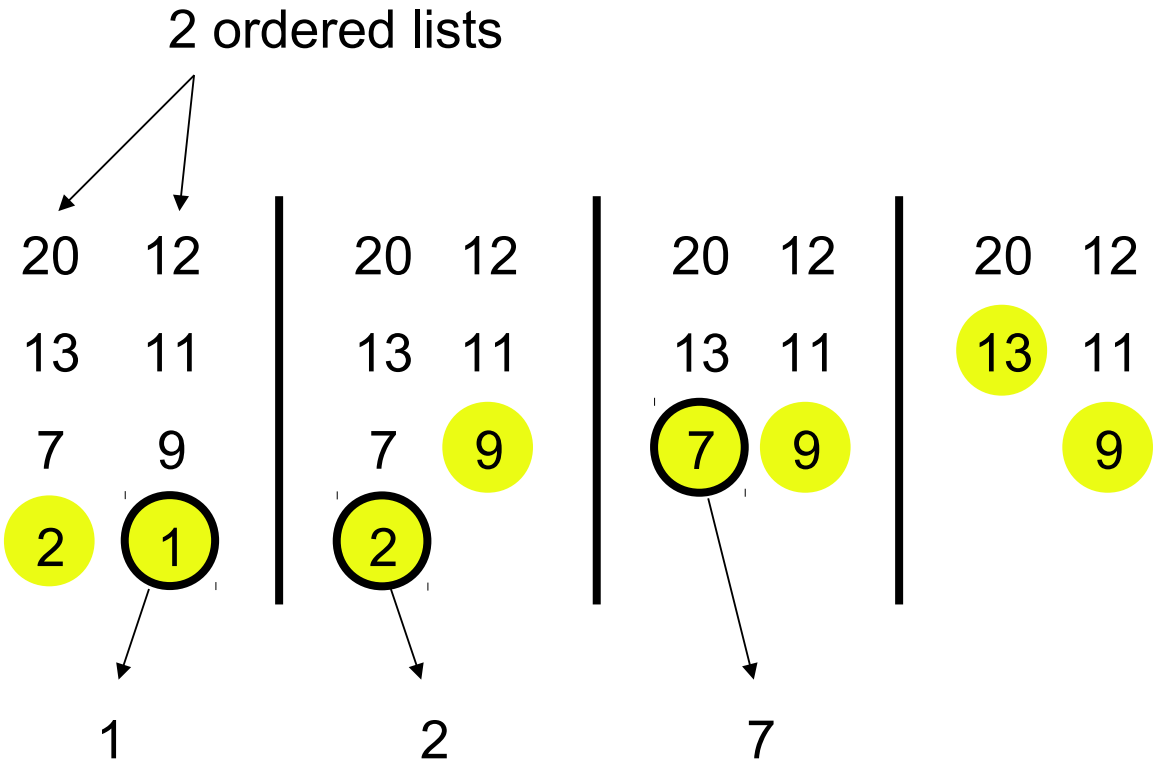
Strategy: examine front of both lists, repeatedly remove smallest

Merge 2 ordered lists (7)



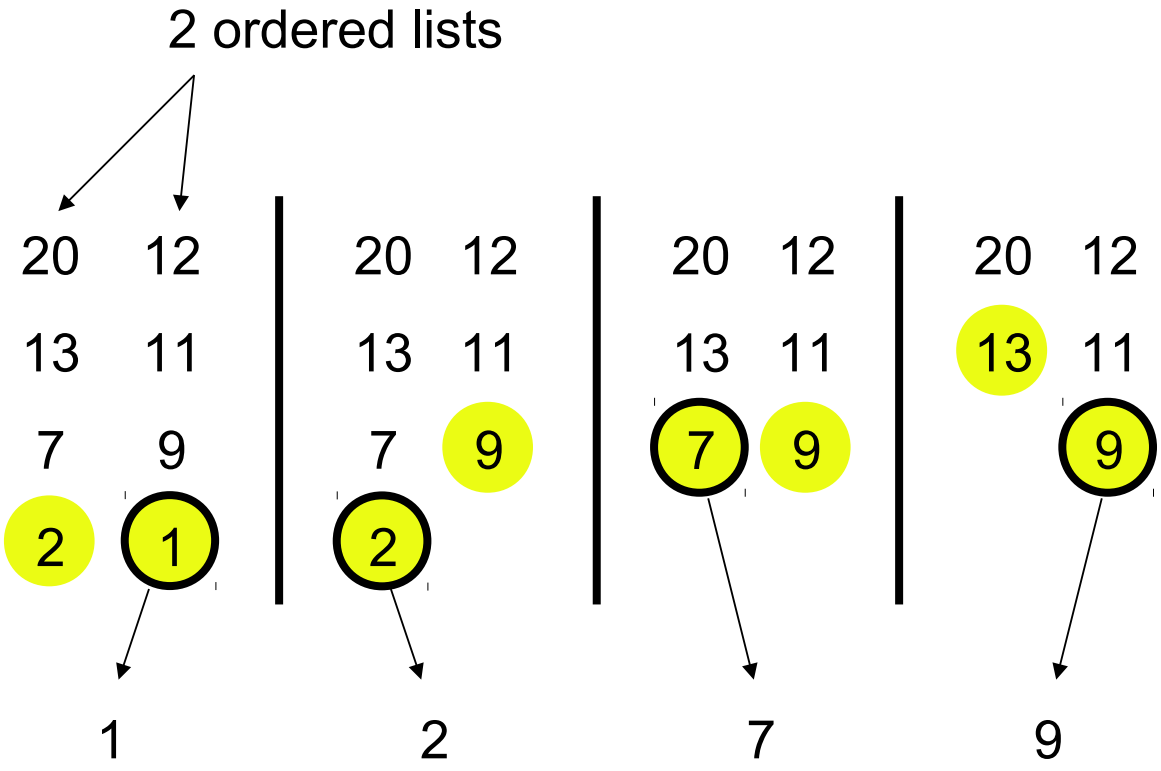
Strategy: examine front of both lists, repeatedly remove smallest

Merge 2 ordered lists (8)



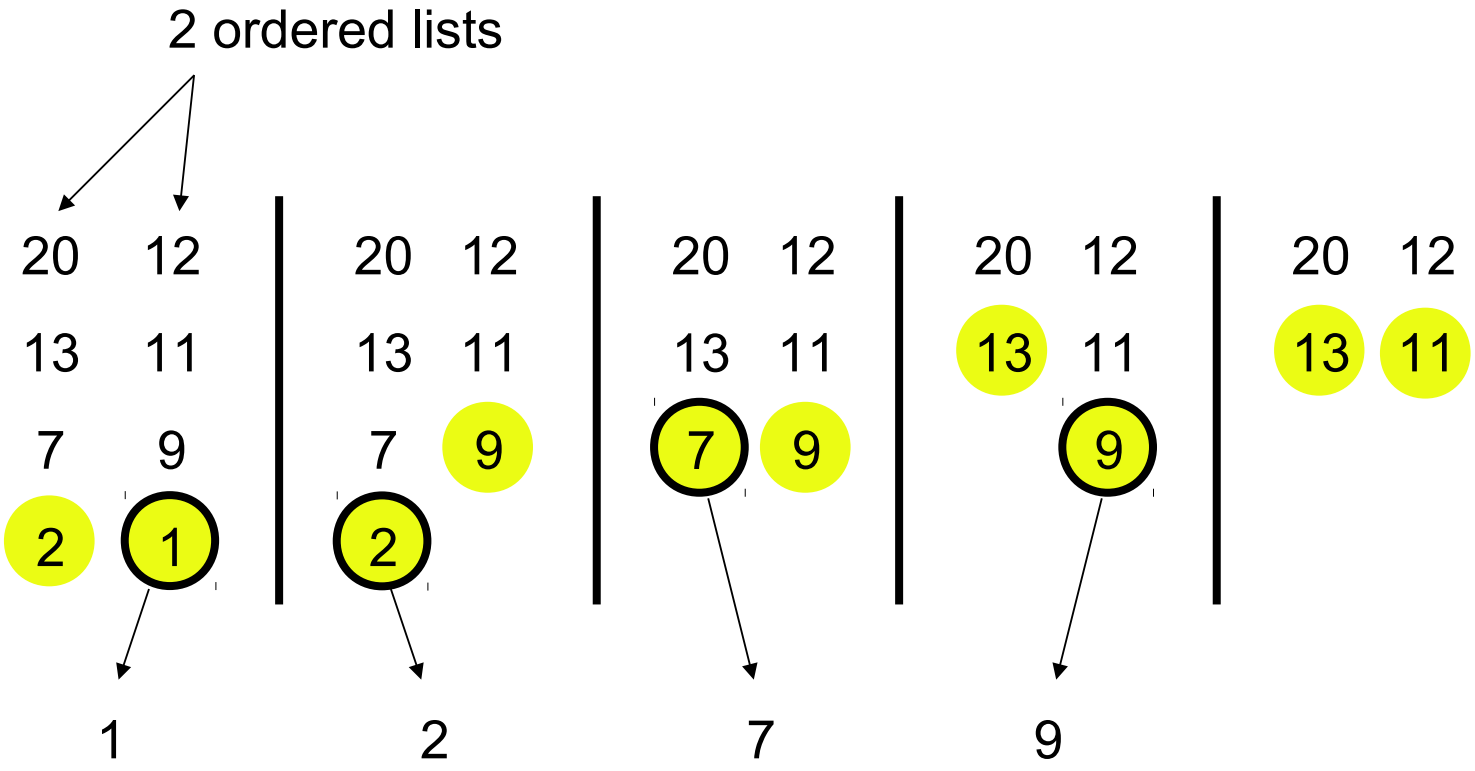
Strategy: examine front of both lists, repeatedly remove smallest

Merge 2 ordered lists (9)



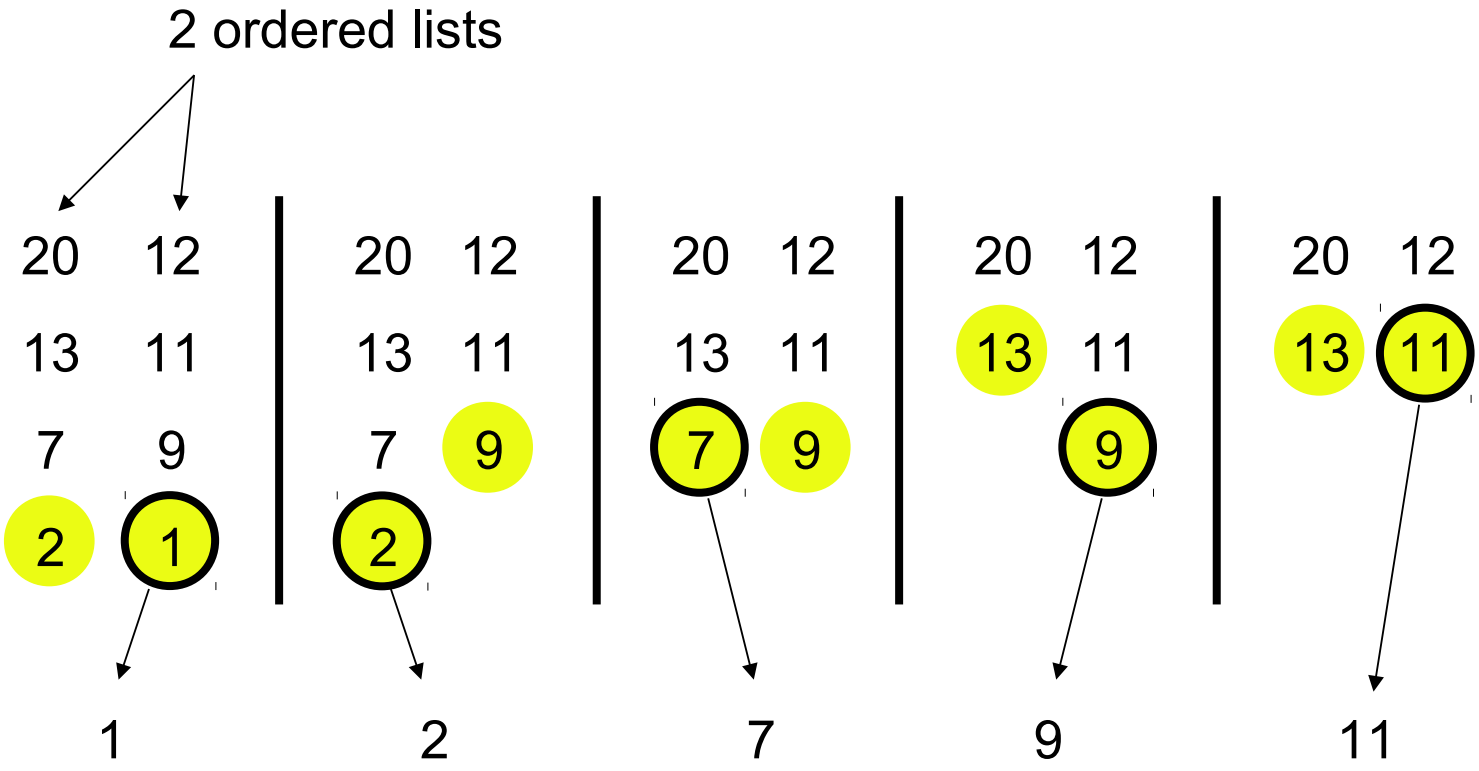
Strategy: examine front of both lists, repeatedly remove smallest

Merge 2 ordered lists (10)



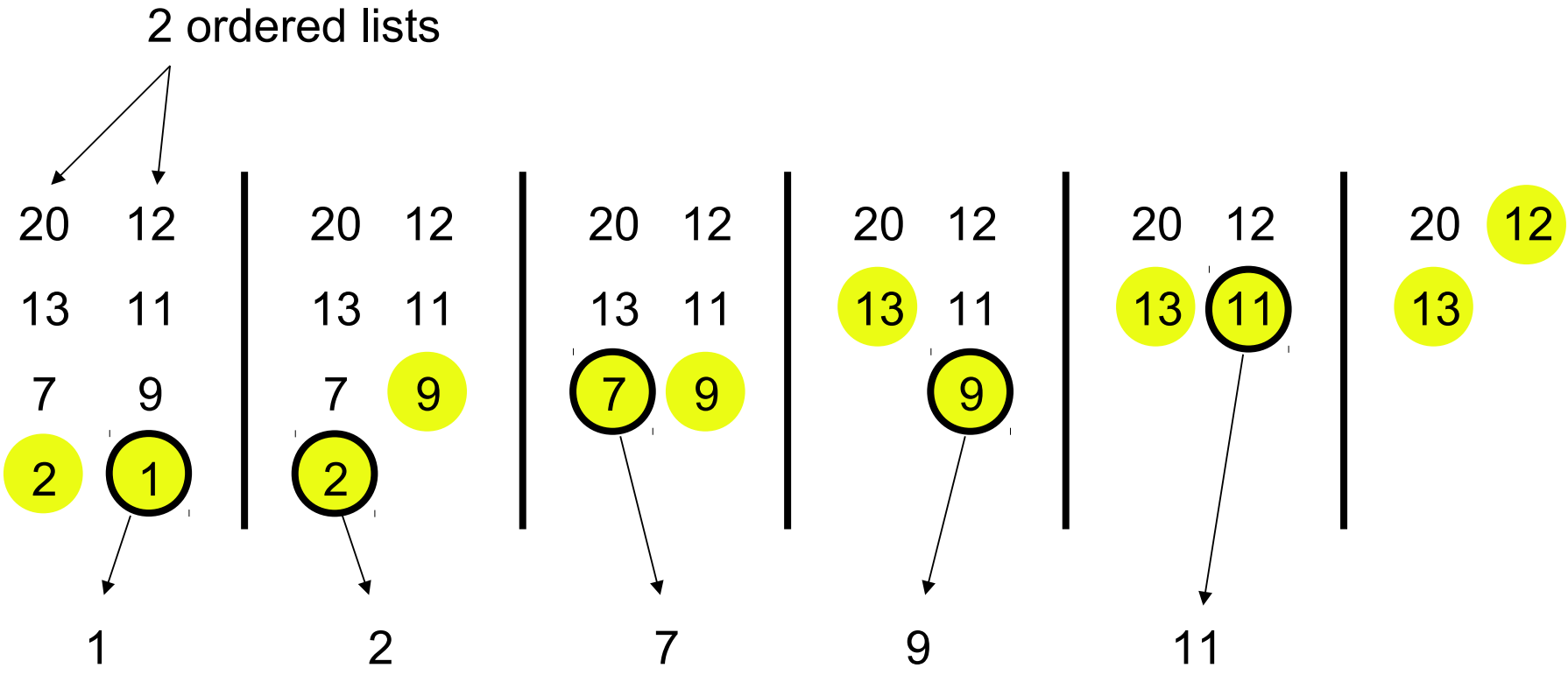
Strategy: examine front of both lists, repeatedly remove smallest

Merge 2 ordered lists (11)



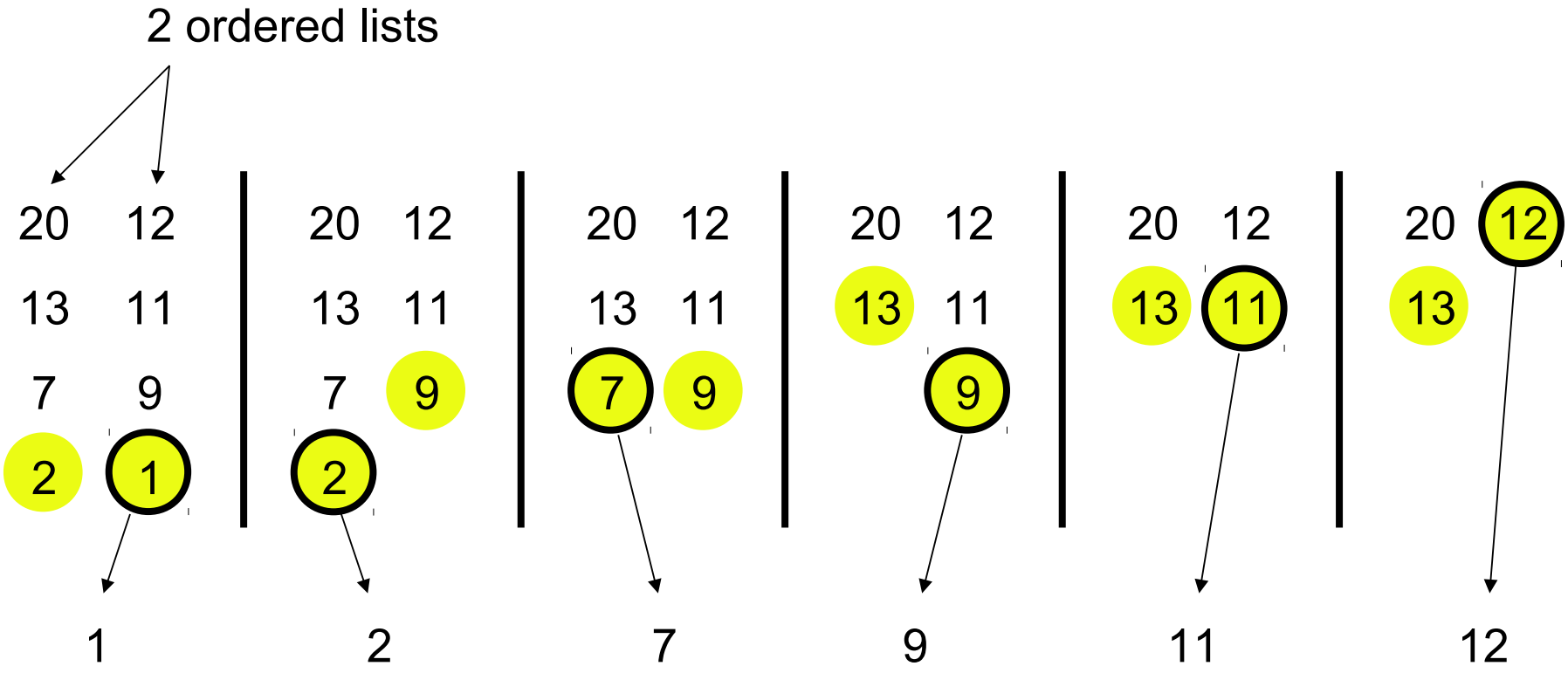
Strategy: examine front of both lists, repeatedly remove smallest

Merge 2 ordered lists (12)



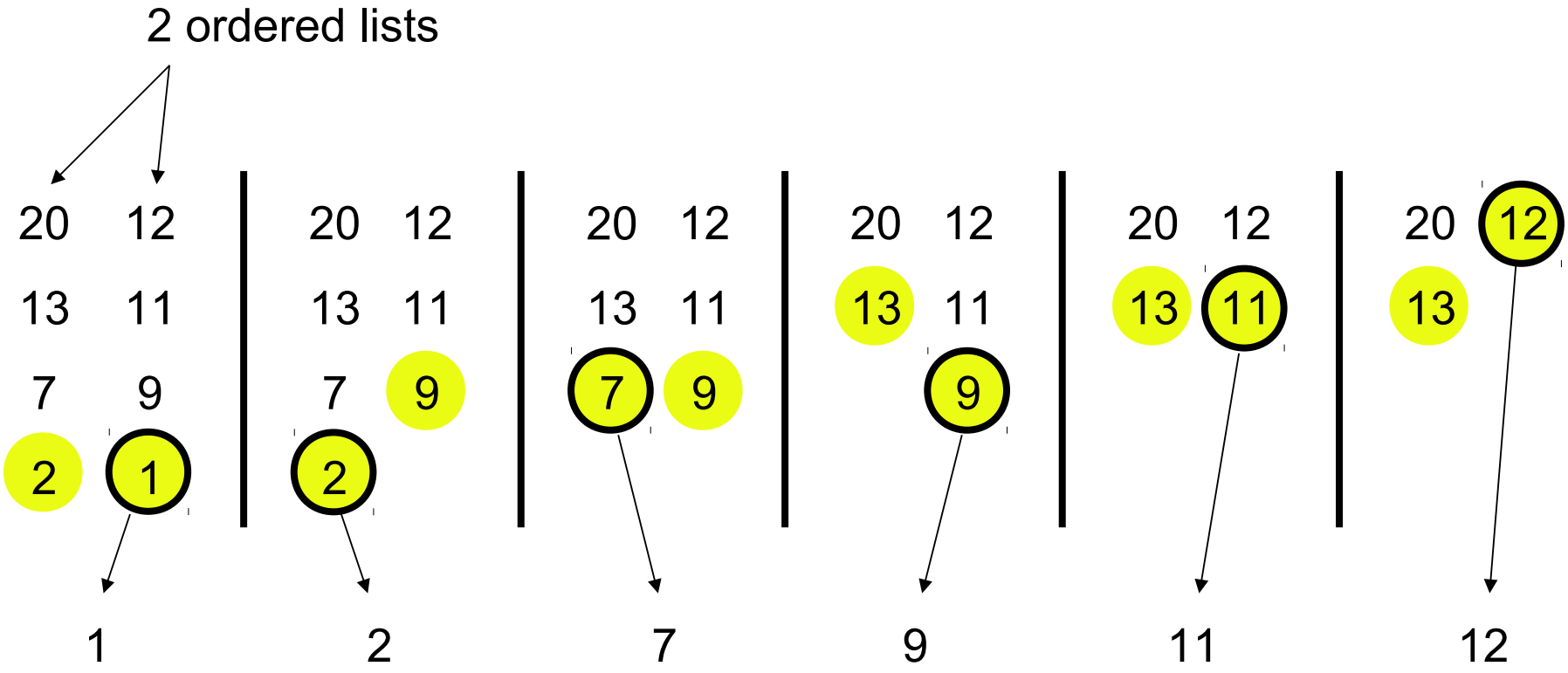
Strategy: examine front of both lists, repeatedly remove smallest

Merge 2 ordered lists (13)



Strategy: examine front of both lists, repeatedly remove smallest

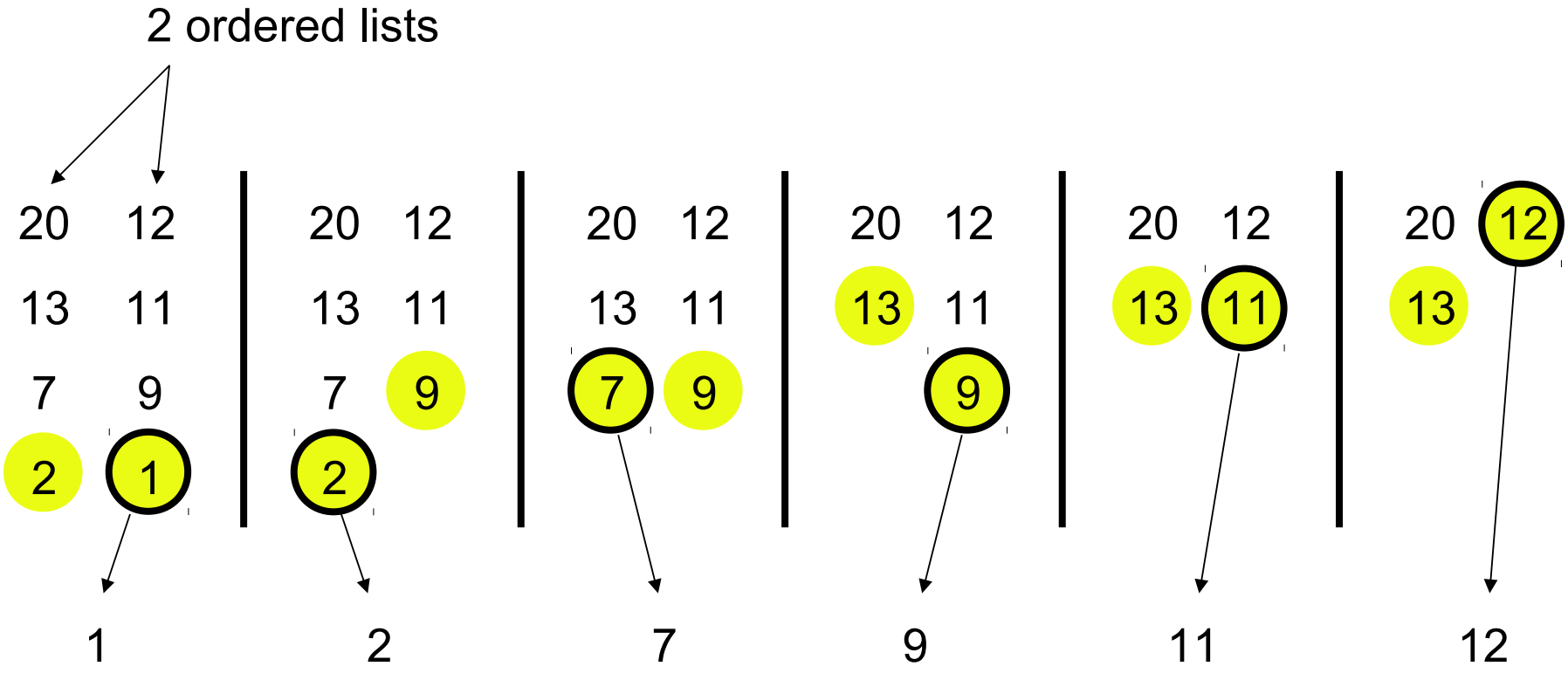
Merge 2 ordered lists (15)



Strategy: examine front of both lists, repeatedly remove smallest

Q: How many comparisons? Complexity of merging?

Merge 2 ordered lists (16)



Strategy: examine front of both lists, repeatedly remove smallest

n steps for n items $= \Theta(n)$

Merge sort asymptotic analysis (1)

```
Function MergeSort(A[1:n]:array of elements)
1. if n = 1 then
2.    return A
3. else
4.    A1 = MergeSort(A[1..n/2])
5.    A2 = MergeSort(A[n/2..n])
6.    return merge(A1, A2)
7. endif
```

So merging requires $O(n)$ work

How about the recurrence relation?

Merge sort asymptotic analysis (2)

```
Function MergeSort(A[1:n]:array of elements)
1. if n = 1 then
2.    return A
3. else
4.    A1 = MergeSort(A[1..n/2])
5.    A2 = MergeSort(A[n/2..n])
6.    return merge(A1, A2)
7. endif
```

Base case: $\Theta(1)$

Recursive case:

$T(n/2)$

$T(n/2)$

$\Theta(n)$

Recurrence relation: $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$

As for Towers of Hanoi, we'll solve using a recursion tree

Merge sort asymptotic analysis (3)

```
Function MergeSort(A[1:n]:array of elements)
1. if n = 1 then
2.    return A
3. else
4.    A1 = MergeSort(A[1..n/2])
5.    A2 = MergeSort(A[n/2..n])
6.    return merge(A1, A2)
7. endif
```

Base case: $\Theta(1)$

Recursive case:
$T(n/2)$
$T(n/2)$
$\Theta(n)$

Recurrence relation: $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$

As for Towers of Hanoi, we'll solve using a recursion tree

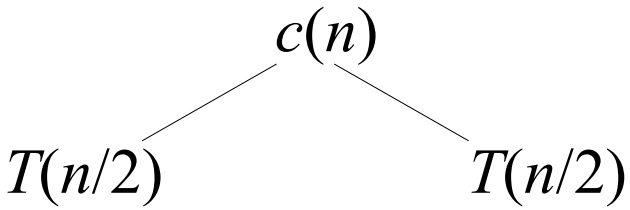
Merge sort asymptotic analysis (4)

Solve: $T(n) = 2\ T(n/2) + cn$

$T(n)$

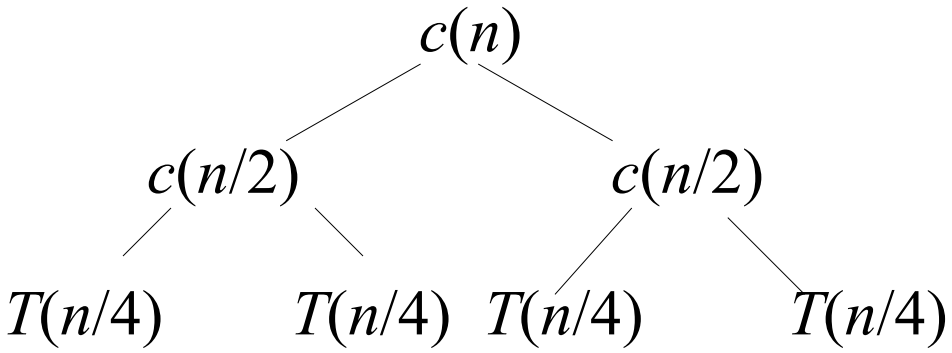
Merge sort asymptotic analysis (5)

Solve: $T(n) = 2\ T(n/2) + cn$



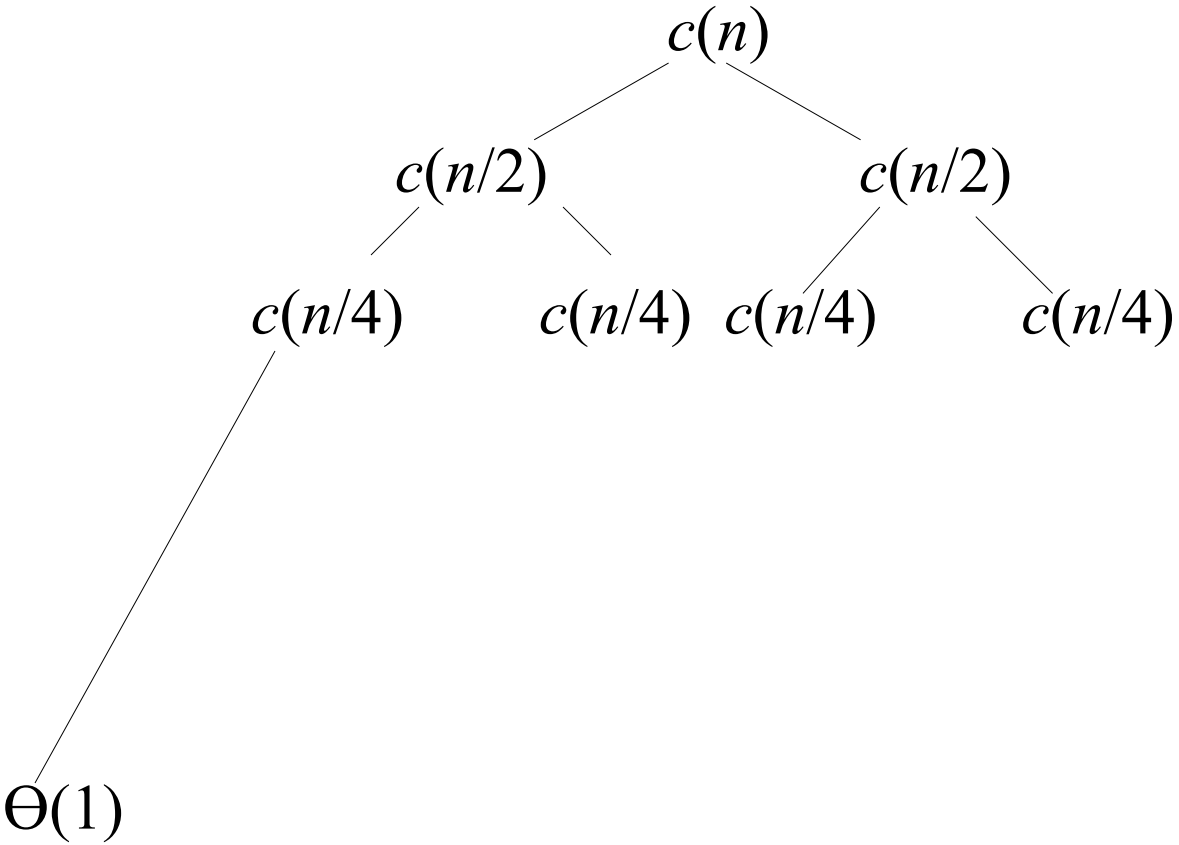
Merge sort asymptotic analysis (6)

Solve: $T(n) = 2\ T(n/2) + cn$



Merge sort asymptotic analysis (7)

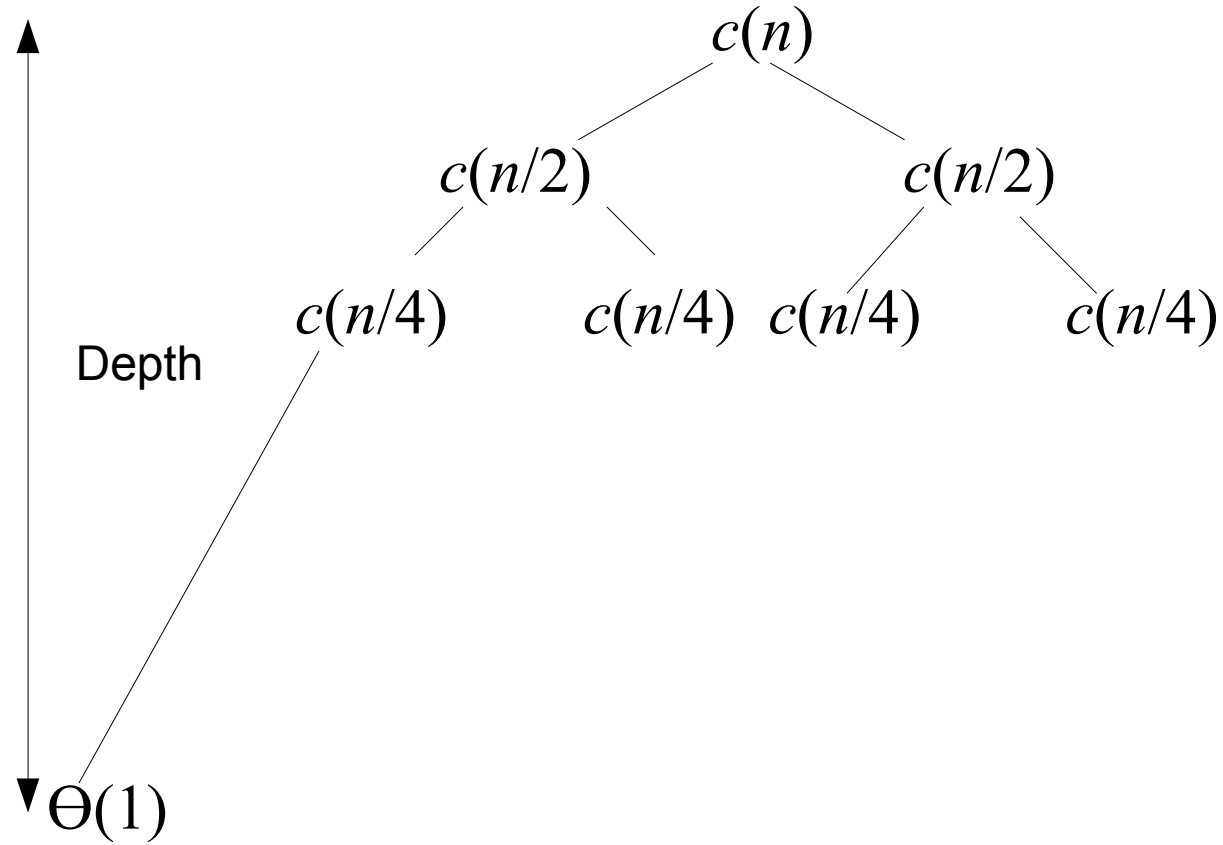
Solve: $T(n) = 2\ T(n/2) + cn$



Merge sort asymptotic analysis (8)

Solve: $T(n) = 2\ T(n/2) + cn$

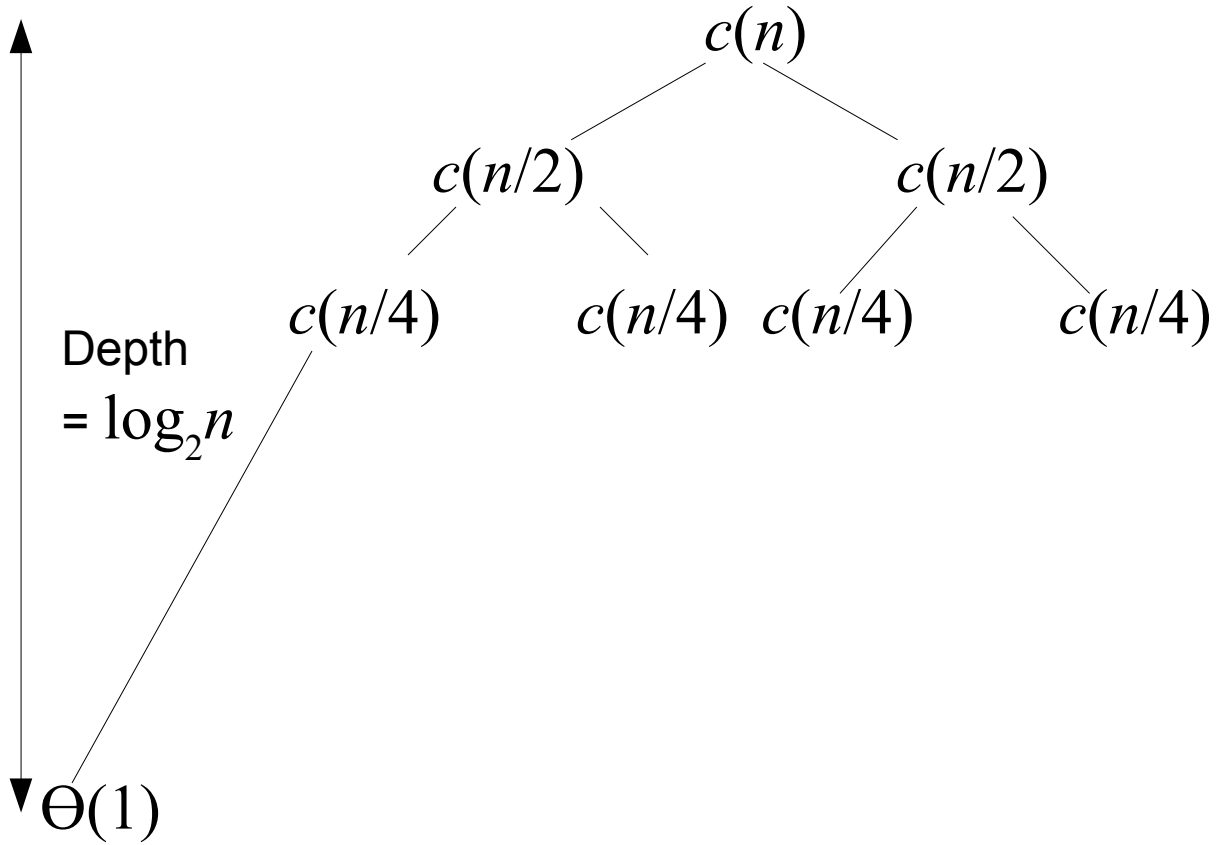
Total work per level?



Merge sort asymptotic analysis (9)

Solve: $T(n) = 2\ T(n/2) + cn$

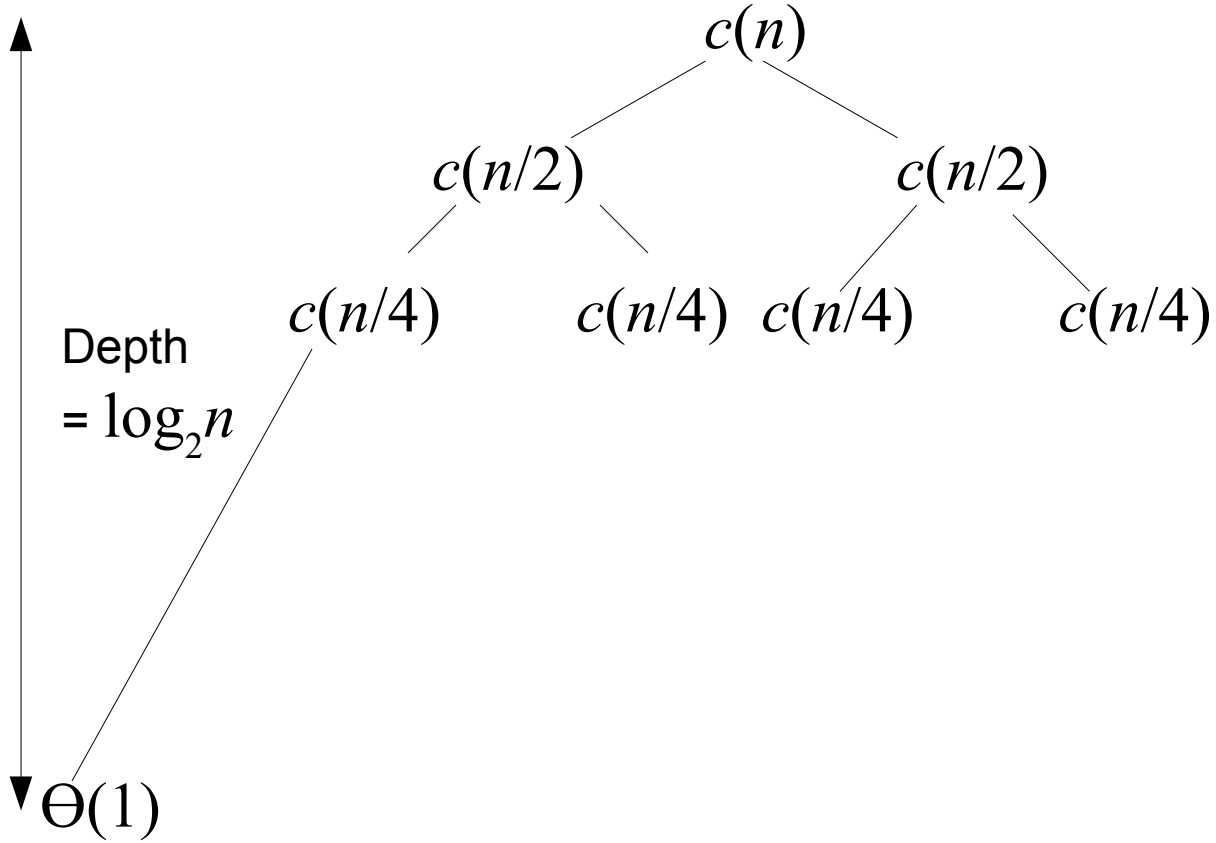
Total work per level?



Merge sort asymptotic analysis (10)

Solve: $T(n) = 2\ T(n/2) + cn$

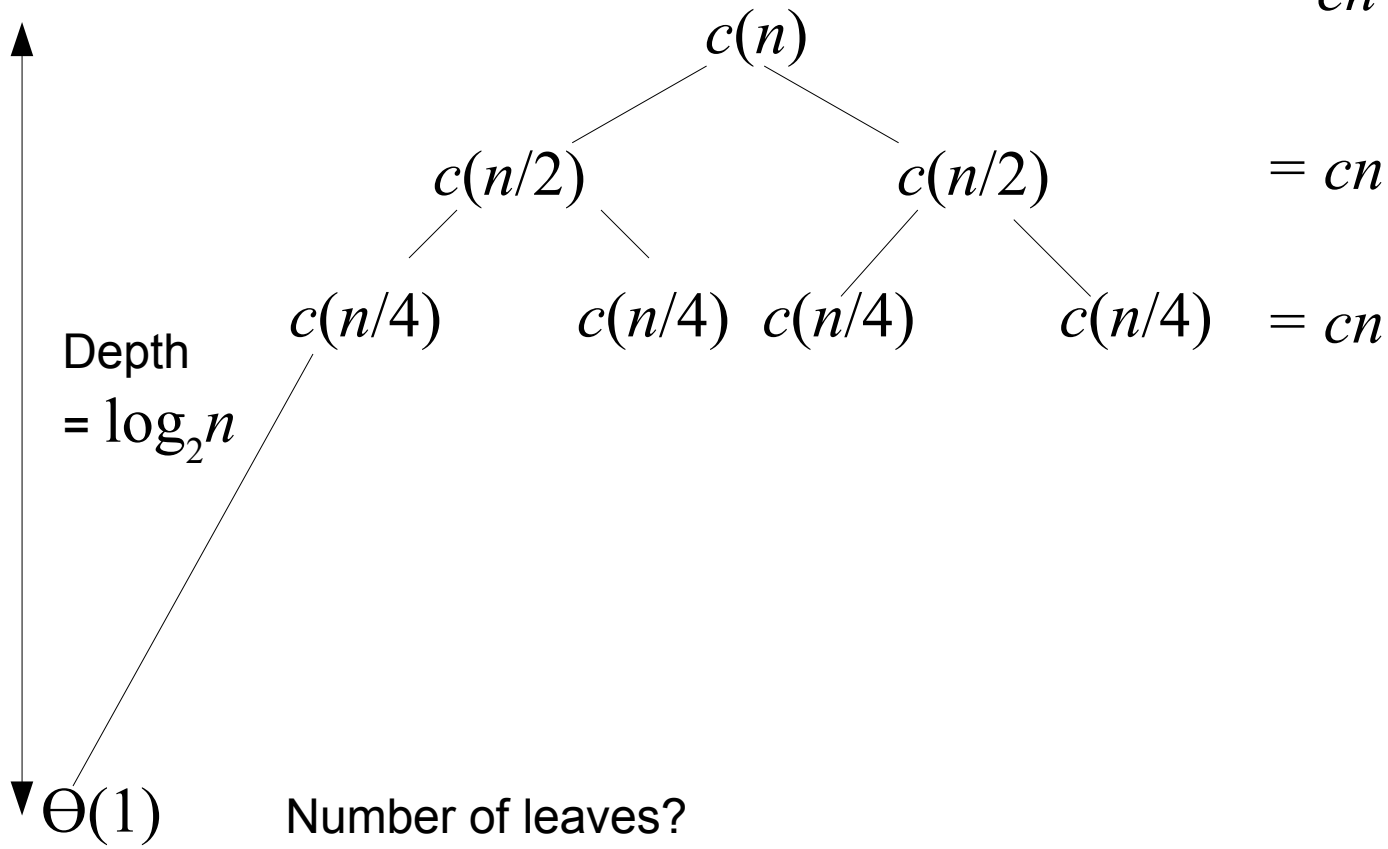
Total work per level?
$= cn$



Merge sort asymptotic analysis (11)

Solve: $T(n) = 2\,T(n/2) + cn$

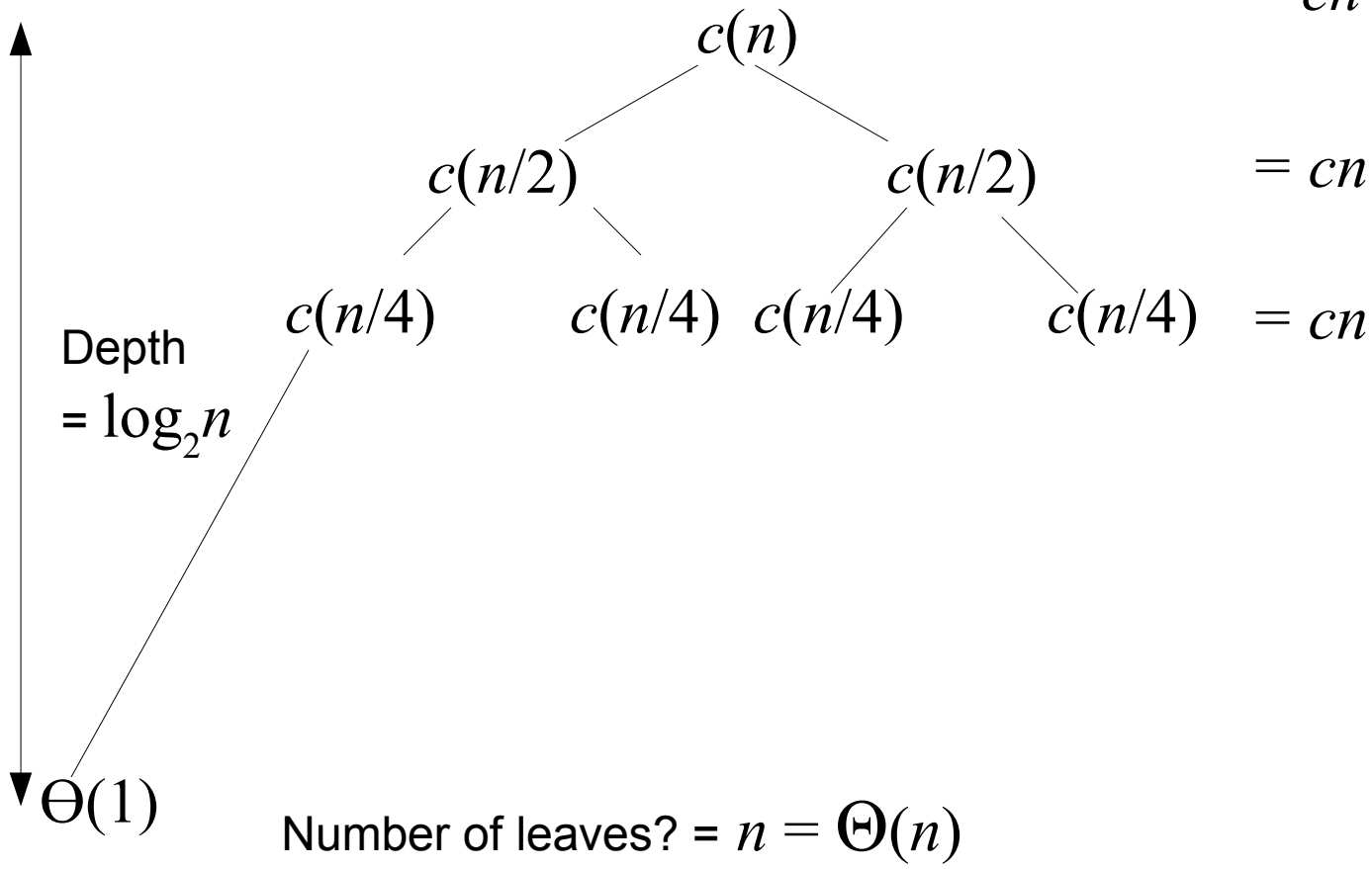
Total work per level?



Merge sort asymptotic analysis (12)

Solve: $T(n) = 2\ T(n/2) + cn$

Total work per level?

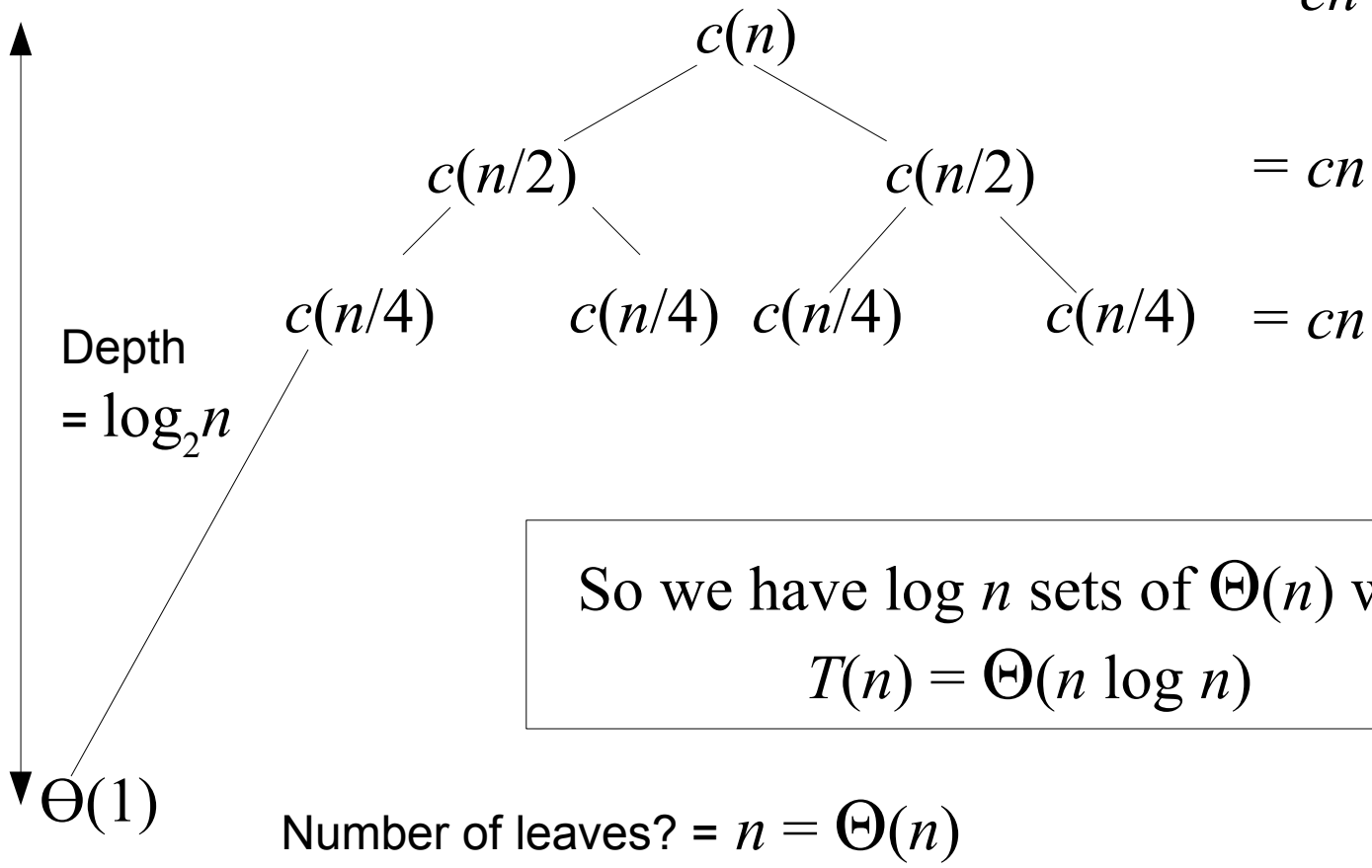


Number of leaves? = $n = \Theta(n)$

Merge sort asymptotic analysis (13)

Solve: $T(n) = 2\ T(n/2) + cn$

Total work per level?



So we have log n sets of $\Theta(n)$ work

$T(n) = \Theta(n \log n)$

Number of leaves? = $n = \Theta(n)$

Merge sort versus Quadratic sort (1)

Conclusion: Merge sort is o(n^2)

Merge sort asymptotically beats insertion sort or selection sort

In practice, merge sort is more efficient than insertion sort approx. when $n > 30$

Merge sort versus Quadratic sort (2)

Conclusion: Merge sort is o(n^2)

Merge sort asymptotically beats insertion sort or selection sort

In practice, merge sort is more efficient than insertion sort approx. when $n > 30$

Obvious question:can we do any better?

Is there a sorting algorithm that is o(n log n)?

Limits of Comparison-based sorting

Think about the algorithms so far

The only way they learn anything about the sequence is by comparing elements

The number of comparisons required to put a set of elements in order is:

$$\Omega(n \log n)$$

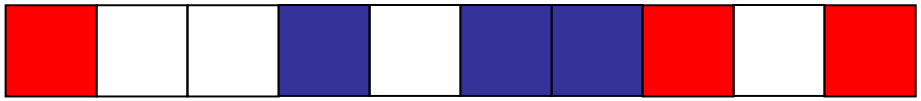
So if our algorithm performs comparisons we can do no better

Q: Do we have to perform comparisons?

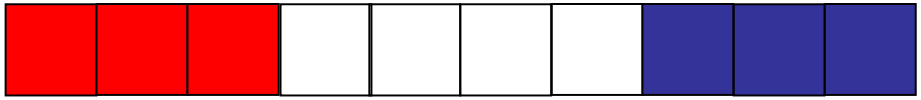
Dutch National Flag (1)

- We're going to finish with a kind of thought experiment
- Consider the following problem:

Input is a sequence of elements which can be one of three colors:

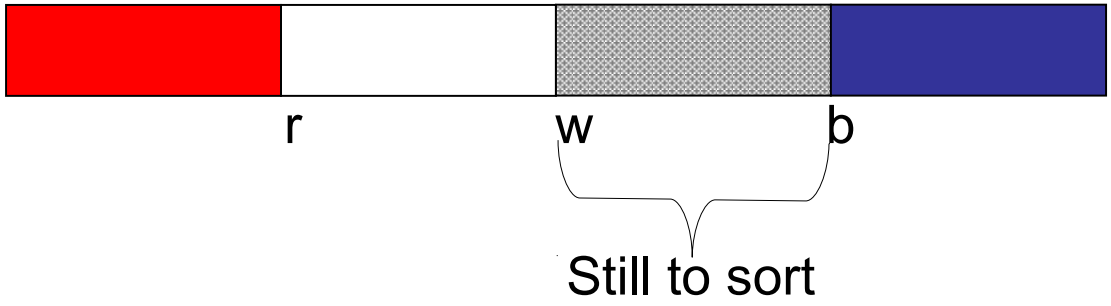


Output is sorted according to the Dutch national flag:



Dutch National Flag (2)

- The algorithm to solve this maintains the following invariant:



Algorithm sketch for n element sequence:

1. Initialize r=1, w=1, b=n+1
2. Repeatedly perform the following until w=b:

   If A[w] = red, swap(A[r], A[w]) and increment r and w

   If A[w] = white, increment w

   If A[w] = blue, swap(A[w], A[b-1]), decrement b

Dutch National Flag (3)

Q: Do you believe this algorithm sorts the flag?

Q:What is the complexity?

Dutch National Flag (4)

Q: Do you believe this algorithm sorts the flag?

Q:What is the complexity?

For each iteration of the loop, we either increment w or decrement b, bringing us one step closer to our termination criteria: w=b

The algorithm is therefore linear in n

We appear to have just seen a $\Theta(n)$ sorting algorithm!

Dutch National Flag (5)

Q: Do you believe this algorithm sorts the flag?

Q:What is the complexity?

For each iteration of the loop, we either increment w or decrement b, bringing us one step closer to our termination criteria: w=b

The algorithm is therefore linear in n

We appear to have just seen a $\Theta(n)$ sorting algorithm!

Q: Any explanation?

Dutch National Flag (6)

Q: Do you believe this algorithm sorts the flag?

Q:What is the complexity?

For each iteration of the loop, we either increment w or decrement b, bringing us one step closer to our termination criteria: w=b

The algorithm is therefore linear in n

We appear to have just seen a $\Theta(n)$ sorting algorithm!

Q: Any explanation?

A:If we know where to put elements by just looking at them, we can sort them in linear time.

Similar idea can be used to any number of potential elements, putting them into “buckets” according to their value

We must know the potential values beforehand

Conclusion

- We've seen some important sorting algorithms:
  - Selection sort and insertion sort (asymptotically quadratic – with insertion sort being better in practice)
  - Merge sort – an elegant application of divide and conquer giving us an O(n log n) sorting algorithm
  - But we've seen when we know in advance what values we might get, we can have linear performance