COMP2823

Lecture 10: Divide and Conquer [GT 3.1 and 8]

A/Prof Julian Mestre
School of Computer Science

Some content is taken from material provided by the textbook publisher Wiley.

THE UNIVERSITY OF SYDNEY

Divide and Conquer

Divide and Conquer algorithms can normally be broken into these three parts:

1. Divide If it is a base case, solve directly, otherwise break up the problem into several parts.

2. Recur/Delegate Recursively solve each part [each sub-problem].

3. Conquer Combine the solutions of each part into the overall solution.

Divide and Conquer

1. Divide If it is a base case, solve directly, otherwise break up the problem into several parts.

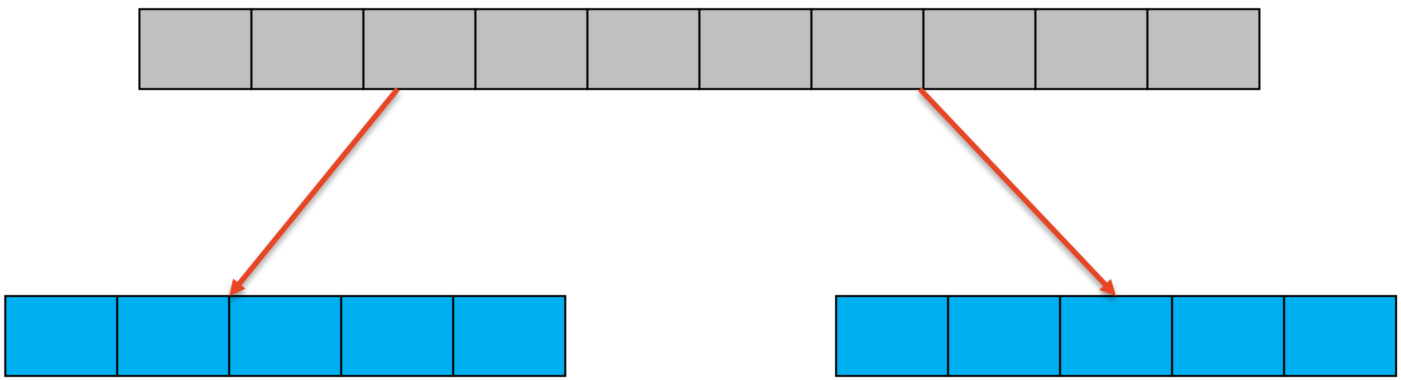
Typical base case:

Subproblem of constant size (usually 0 or 1 elements) for which you can compute the solution explicitly

easy to compute solution

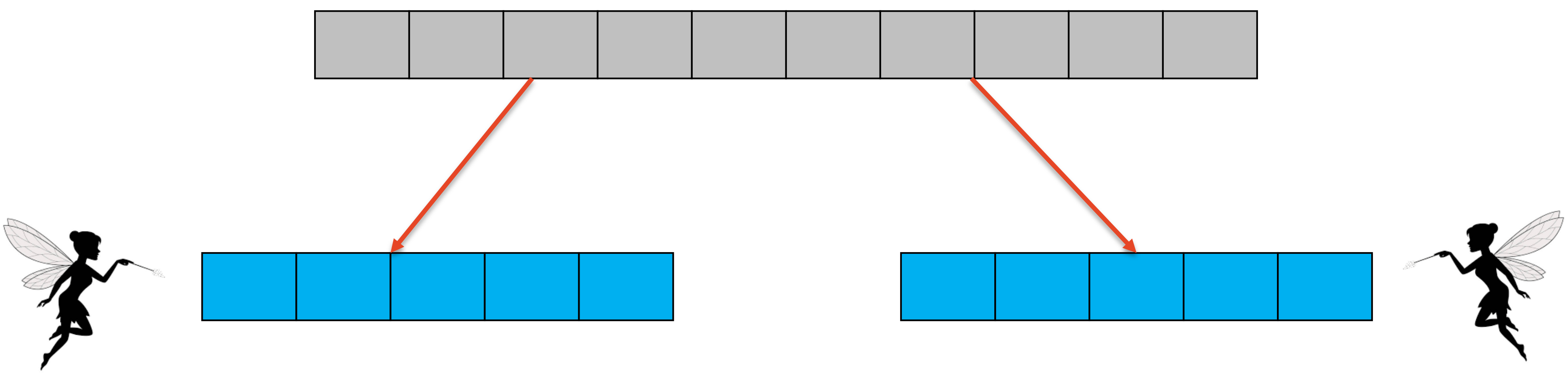
Divide and Conquer

1. Divide If it is a base case, solve directly, otherwise break up the problem into several parts.



Divide and Conquer

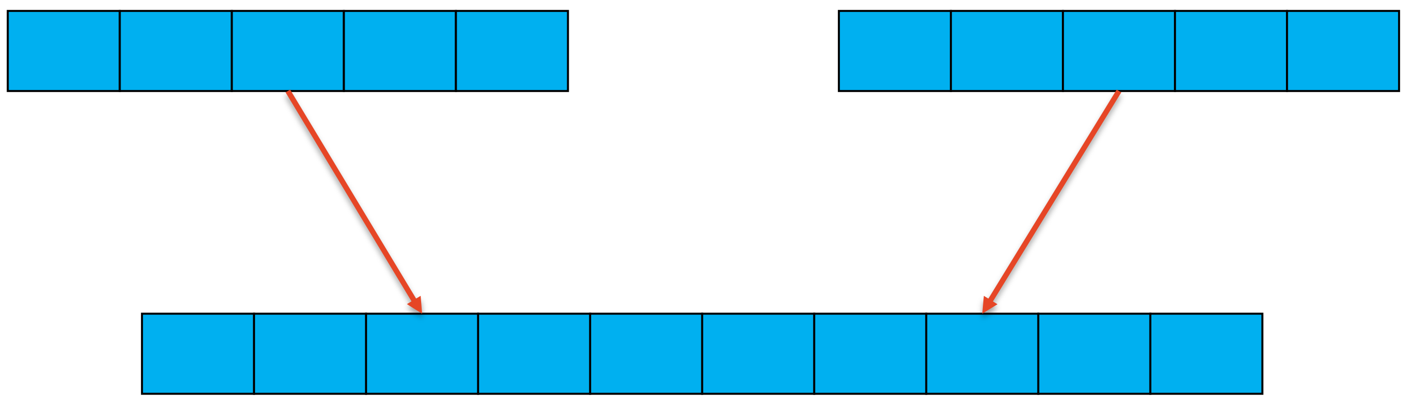
2. Recur/Delegate Recursively solve each part [each sub-problem].



The sub-problems are solved by the Recursion Fairy (similar to induction hypothesis), so we don't have to worry about them.

Divide and Conquer

3. Conquer Combine the solutions of each part into the overall solution.



Searching Sorted Array

Given A sorted sequence S of n numbers $a_0, a_1, \ldots, a_{n-1}$ stored in an array A[0, 1, …, n - 1].

Problem Given a number x, is x in S?

0	2	5	7	12	22	25	37	39	50	55	80

Searching: Naïve Approach

Problem Given a number x, is x in S?

Idea Check every element in turn to see if it is equal to x.

```
for e in S do
   if e equals x then
      return "Yes"
return "No"
```

Found an element equal to x in S

There was no element equal to x in S

0	2	5	7	12	22	25	37	39	50	55	80

Running Time O(n)

Binary Search in sorted A[0 to n-1]

1. If the array is empty, then return "No"
2. Compare x to the middle element, namely $A[\lfloor n/2 \rfloor]$
3. If this middle element is x, then return "Yes"
4. When the middle element is not x: if $A[\lfloor n/2 \rfloor] > x$, then recursively search $A[0{:}\lfloor n/2 \rfloor]$
5. if $A[\lfloor n/2 \rfloor] < x$, then recursively search $A[\lfloor n/2 \rfloor + 1{:}n]$

0	2	5	7	12	22	25	37	39	50	55	80

Heads up: pseudocode textbook uses indexing from 1 to n, not 0 to n-1

Binary Search Pseudocode

```
def binary_search(A, left, right, x):
   # A is sorted and left <= right
   # looking for x in A[left:right]

   if left = right then
      return "unsuccessful"

   mid = floor((left + right) / 2)
   if A[mid] < x then
      return binary_search(A, mid + 1, right, x)
   else if A[mid] > x then
      return binary_search(A, left, mid, x)
   else
      return mid
```

We can't do ceiling,
it doesn't work for, e.g.:
left_idx = 10
right_idx = 11

mid = 10.5 -> 11 if do ceiling
ceiling will be testing outside the range

Heads up: pseudocode textbook uses indexing from 1 to n, not 0 to n-1

Binary Search

– Example, search for x=5

0	2	5	7	12	22	25	37	39	50	55	80

Binary Search

– Example, search for x=5

0	2	5	7	12	22	25	37	39	50	55	80

A[6]

Binary Search

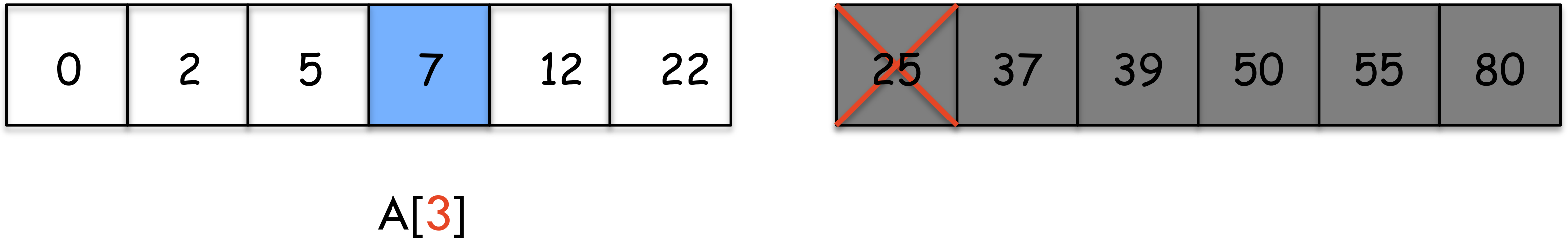
- Example, search for x=5

0	2	5	7	12	22	25	37	39	50	55	80

A[6] $= 25 > 5 = x$

Binary Search

- Example, search for x=5



Binary Search

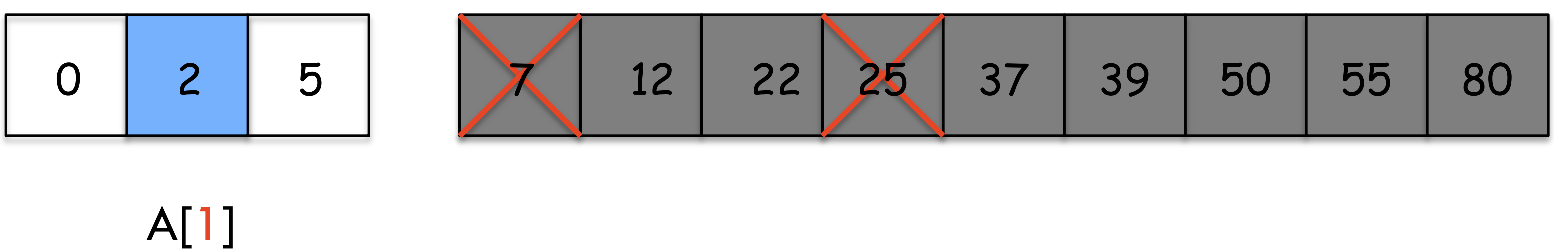
– Example, search for x=5

0	2	5	7	12	22	~~25~~	37	39	50	55	80

$\text{A}[3] = 7 > 5 = x$

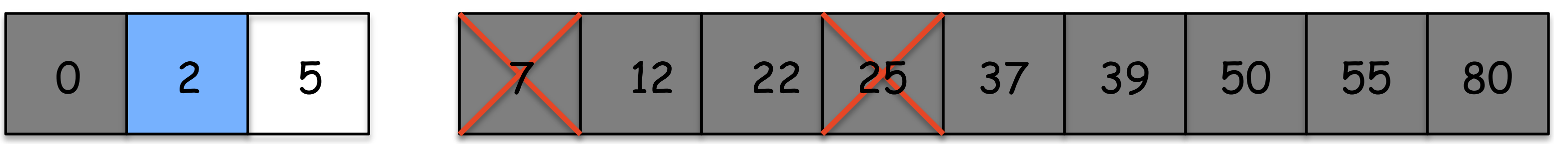
Binary Search

– Example, search for x=5



Binary Search

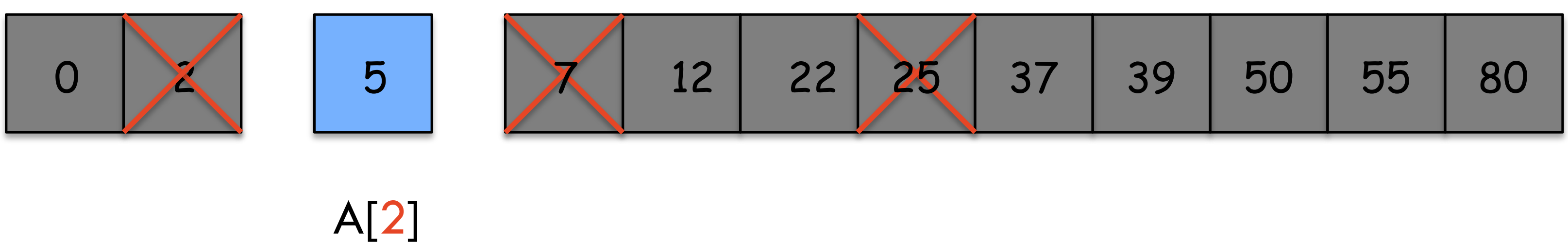
- Example, search for x=5



A[1] $= 2 < 5 = x$

Binary Search

– Example, search for x=5



Binary search correctness

Proof by invariant

Invariant: If x is in A before the divide step, then x is in A after the divide step

- if $A[\lfloor n/2 \rfloor] > x$, then x must be in $A[0: \lfloor n/2 \rfloor]$
- if $A[\lfloor n/2 \rfloor] < x$, then x must be in $A[\lfloor n/2 \rfloor + 1 : n]$

Every divide step leads to a smaller array.

Thus, if x in A, we will eventually inspect its position due to the invariant and return "Yes".

Thus, if x in not in A, then eventually we reach the empty array and return "No".

Recurrence formula

An easy way to analyze the time complexity of a divide-and-conquer algorithm is to define and solve a recurrence

Let T(n) be the running time of the algorithm, we need to find out:

- Divide step cost in terms of n
- Recur step(s) cost in terms of T(smaller values)
- Conquer step cost in terms of n

Together with information about the base case, we can set up a recurrence for T(n) and then solve it.

$$T(n) = \begin{cases} \text{“Recur” + “Divide and Conquer”} & \text{for } n > 1 \\ \text{“Base case” (typically } O(1)) & \text{for } n = 1 \end{cases}$$

Binary search on an array complexity analysis

Divide step (find middle and compare to x) takes $O(1)$

Recur step (solve left or right subproblem) takes $T(n/2)$

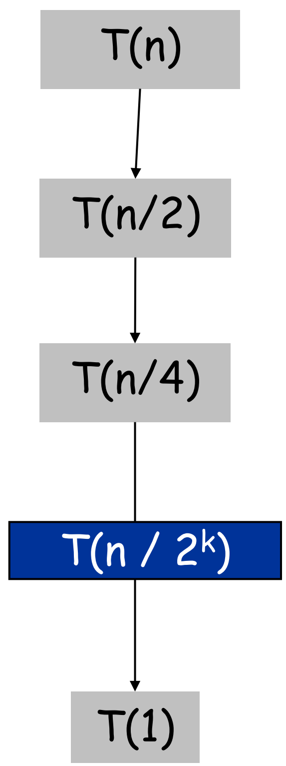
Conquer step (return answer from recursion) takes $O(1)$

Now we can set up the recurrence for $T(n)$:

$$T(n) = \begin{cases} T(n/2) + O(1) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(\log n)$, since we can only halve the input $O(\log n)$ times before reaching a base case

Proof by unrolling: $T(n) = T(n/2) + O(1)$



1 (of size n)

1 (of size n/2)

1 (of size n/4)

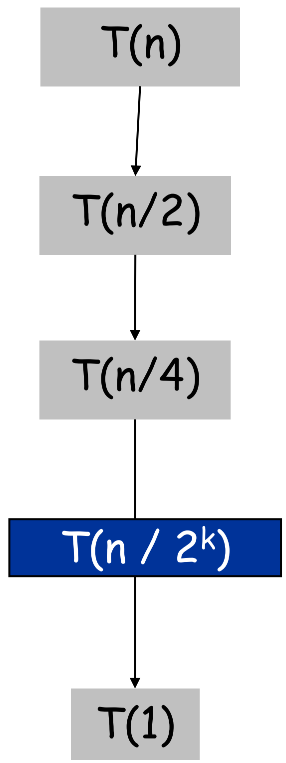
. . .

1 (of size $n/2^k$)

. . .

1 (of size 1)

Proof by unrolling: $T(n) = T(n/2) + O(1)$



$\log_2 n$

1 (of size n) $\rightarrow$ c

1 (of size $n/2$) $\rightarrow$ c

1 (of size $n/4$) $\rightarrow$ c

. . .

1 (of size $n/2^k$) $\rightarrow$ c

. . .

1 (of size 1) $\rightarrow$ c

$c \log_2 n$

Binary search on a linked list complexity analysis

Divide step (find middle and compare to x) takes $O(n)$

Recur step (solve left or right subproblem) takes $T(n/2)$

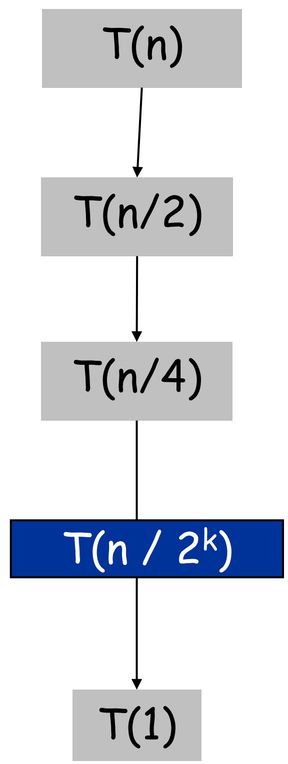
Conquer step (return answer from recursion) takes $O(1)$

Now we can set up the recurrence for $T(n)$:

$$T(n) = \begin{cases} T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n)$, since to access the next index we end up with $n/2 + n/4 + n/8 + \ldots$

Proof by unrolling: T(n) = T(n/2) + O(n)



1 (of size n)

1 (of size n/2)

1 (of size n/4)

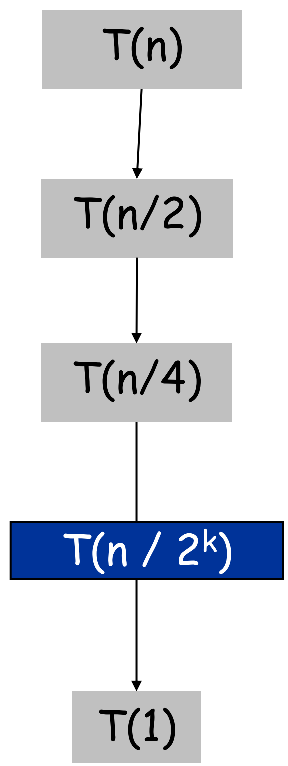
. . .

1 (of size $n/2^k$)

. . .

1 (of size 1)

Proof by unrolling: T(n) = T(n/2) + O(n)



$\log_2 n$

1 (of size n) $\rightarrow$ c n

1 (of size n/2) $\rightarrow$ c n/2

1 (of size n/4) $\rightarrow$ c n/4

. . .

1 (of size $n/2^k$) $\rightarrow$ c $n/2^k$

. . .

1 (of size 1) $\rightarrow$ c 1

---

2 c n

Merge-Sort

1. Divide the array into two halves.
2. Recur recursively sort each half.
3. Conquer two sorted halves to make a single sorted array.

1	12	5	16	19	7	23	6	13	20

Divide

1	12	5	16	19	7	23	6	13	20

Recur

1	5	12	16	19	6	7	13	20	23

Conquer

1	5	6	7	12	13	16	19	20	23

Merge-Sort pseudocode

```
def merge_sort(S):
  # base case
  if |S| < 2 then
    return S

  # divide
  mid ← ⌊|S|/2⌋
  left ← S[:mid]     # doesn't include S[mid]
  right ← S[mid:]    # includes S[mid]


  # recur
  sorted_left ← merge_sort(left)
  sorted_right ← merge_sort(right)

  # conquer
  return merge(sorted_left, sorted_right)
```

How?

Merge

Input Two sorted lists.

Output A new merged sorted list.

To merge, we use:

- O(n) comparisons.
- An array to store our results.

1	5	12	16	19

6	7	13	20	23

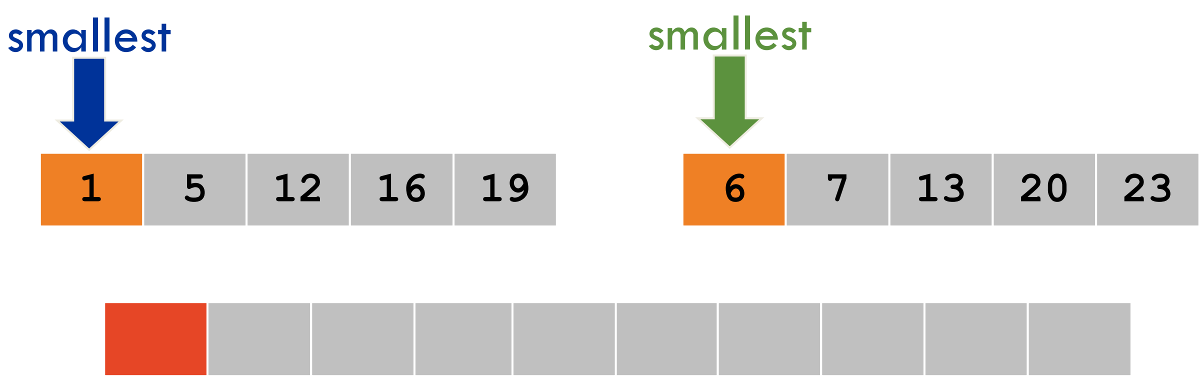
Result:

1	5	6	7	12					

Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.

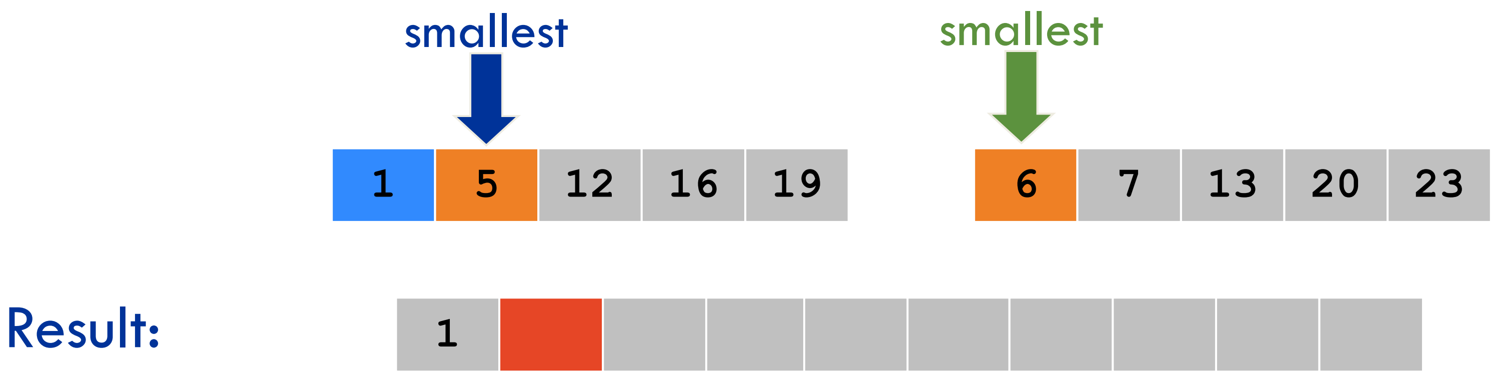


Result:

Merge

Merge Algorithm

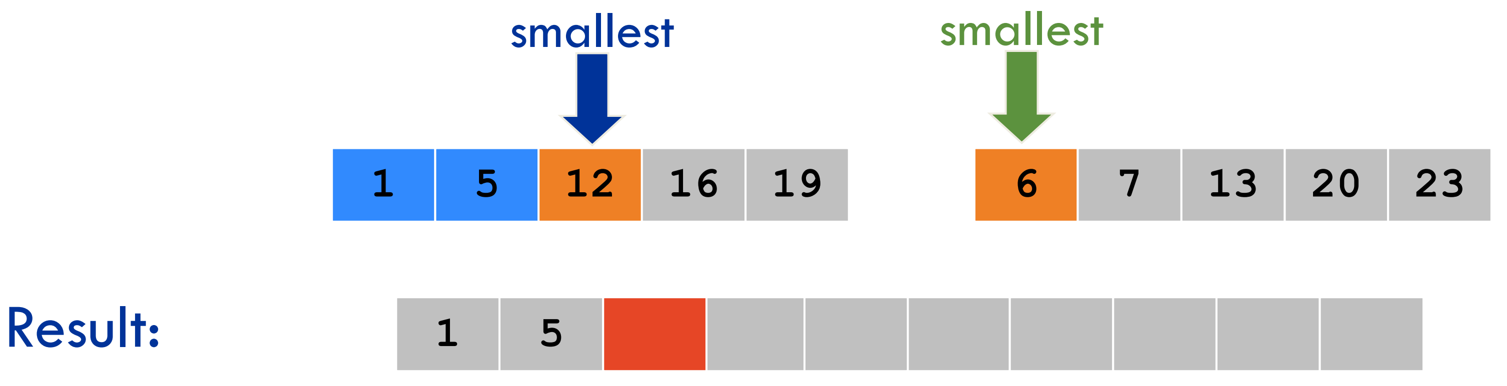
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

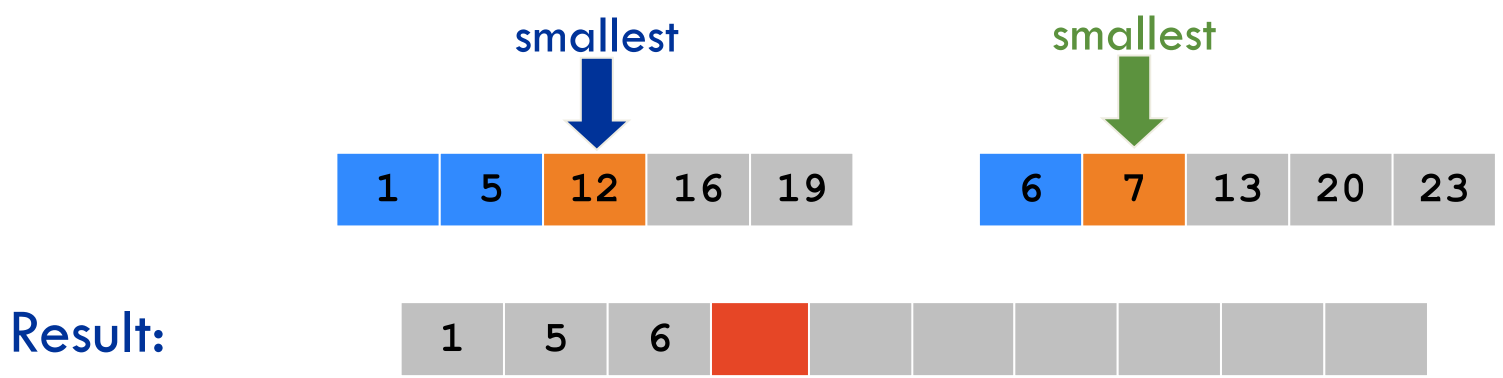
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

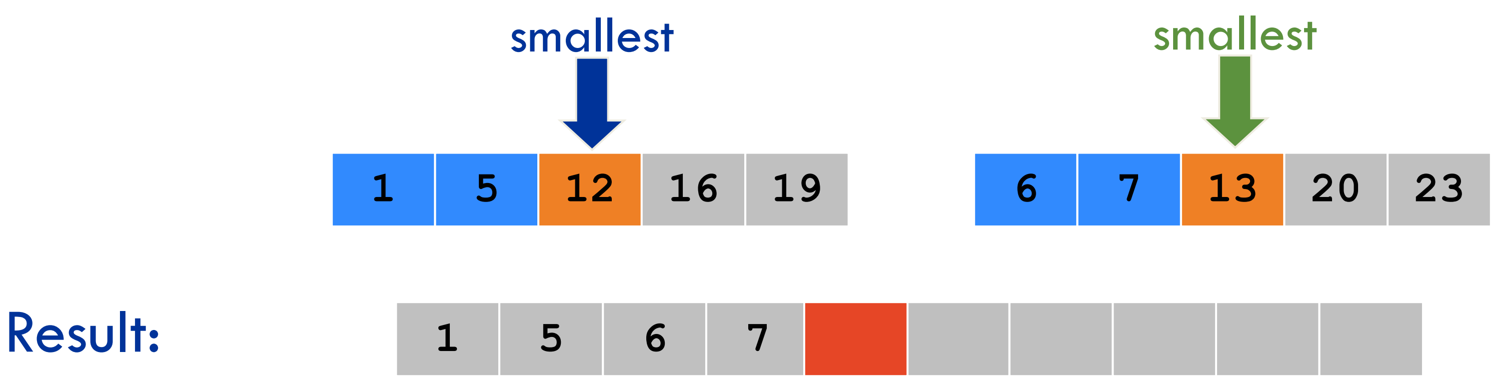
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

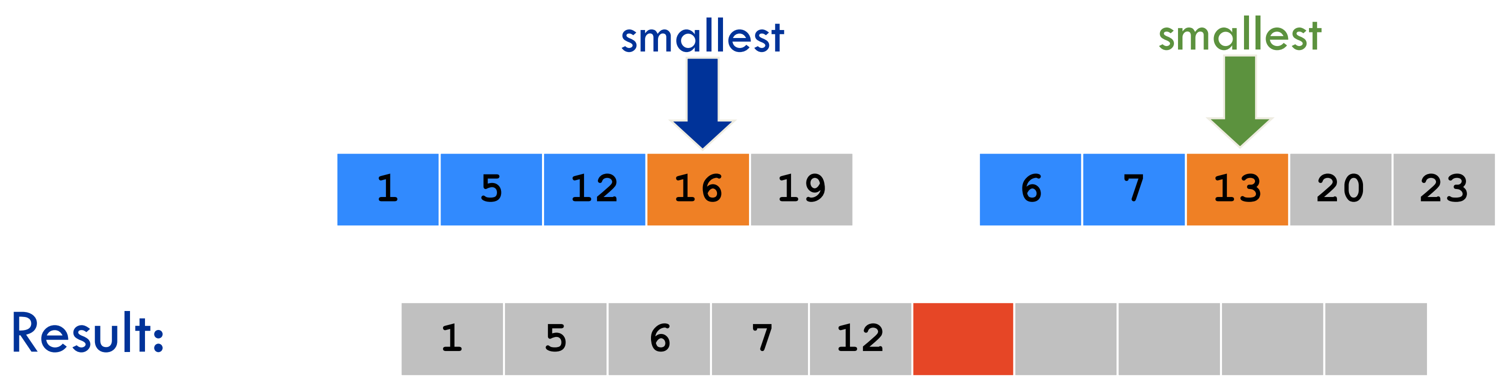
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

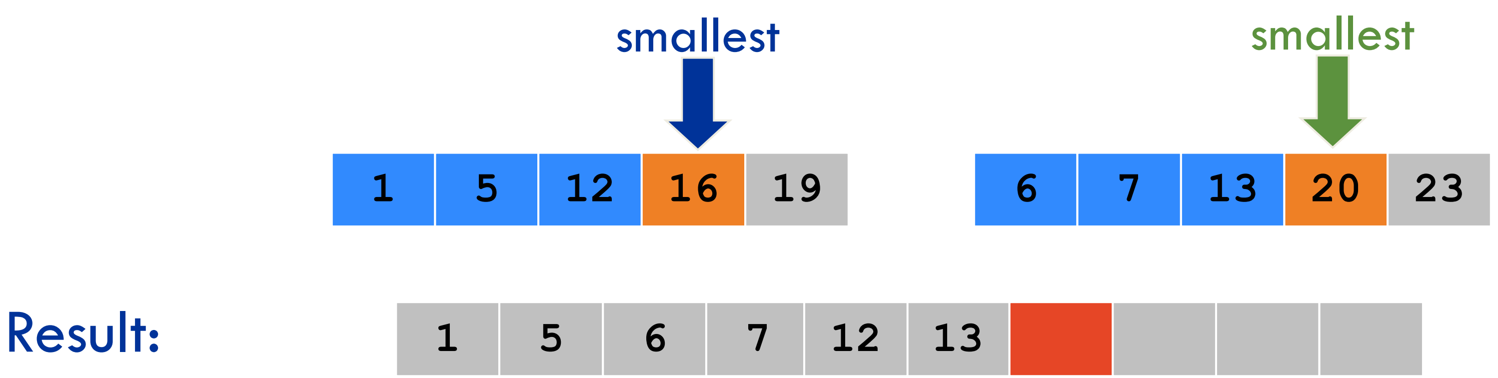
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

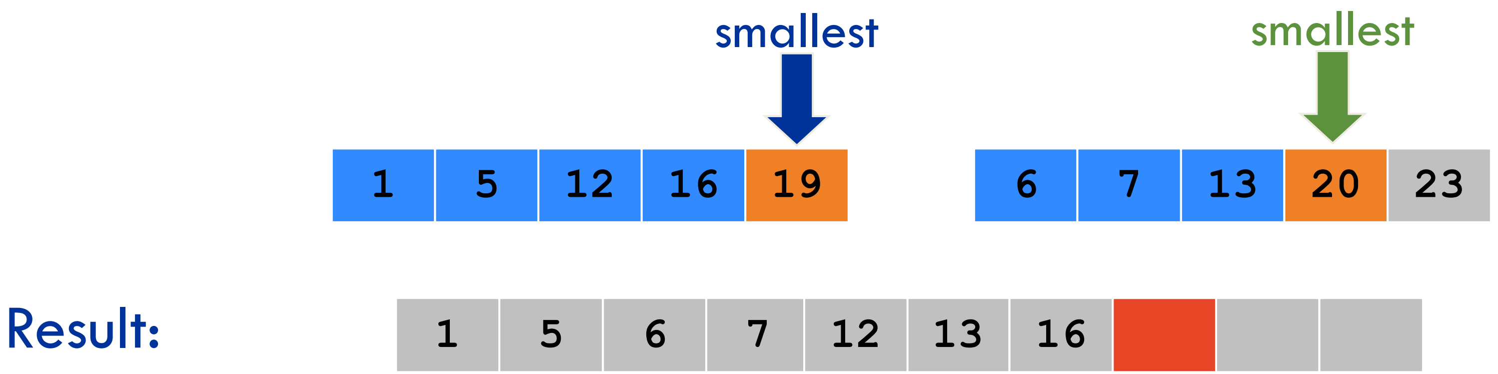
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

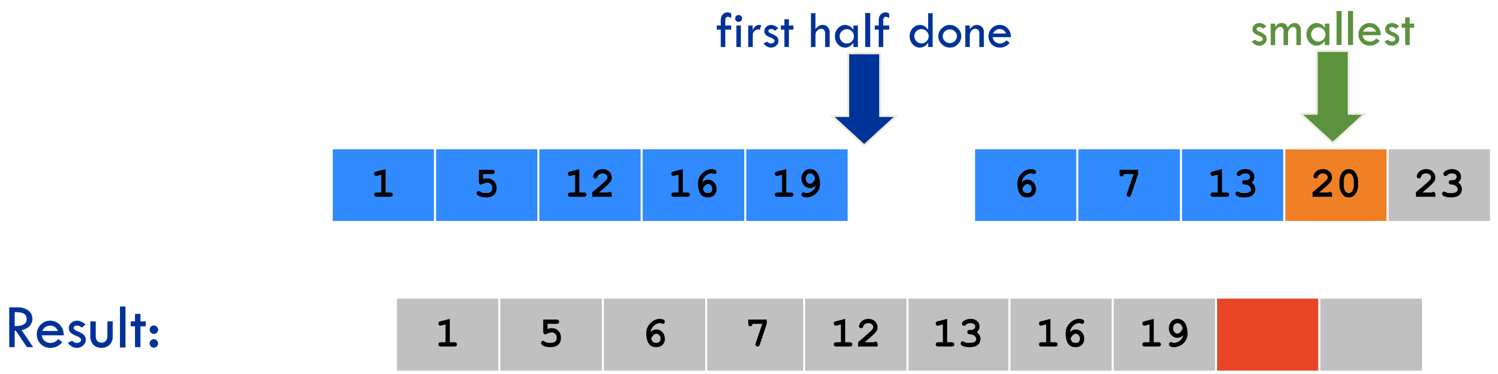
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

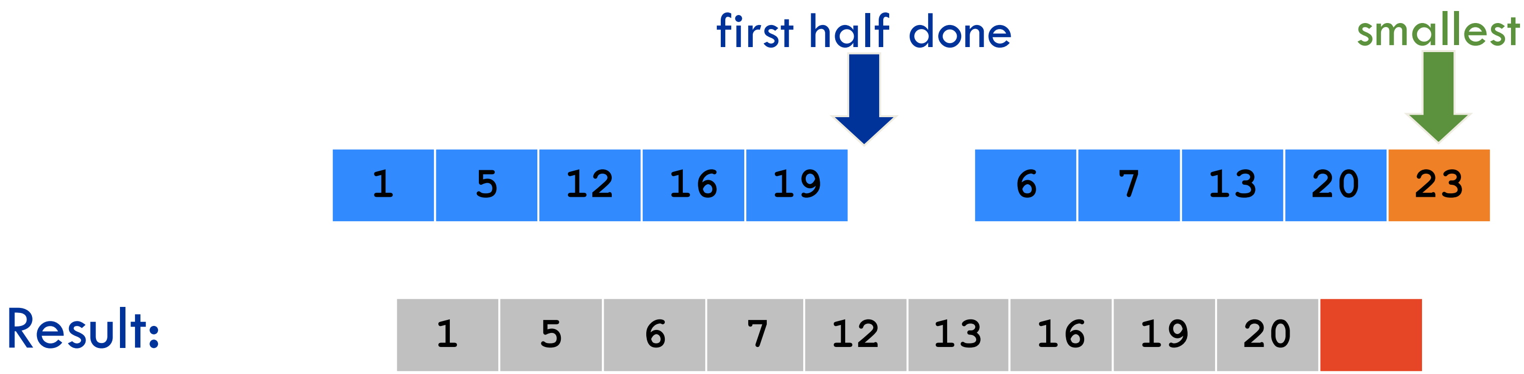
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

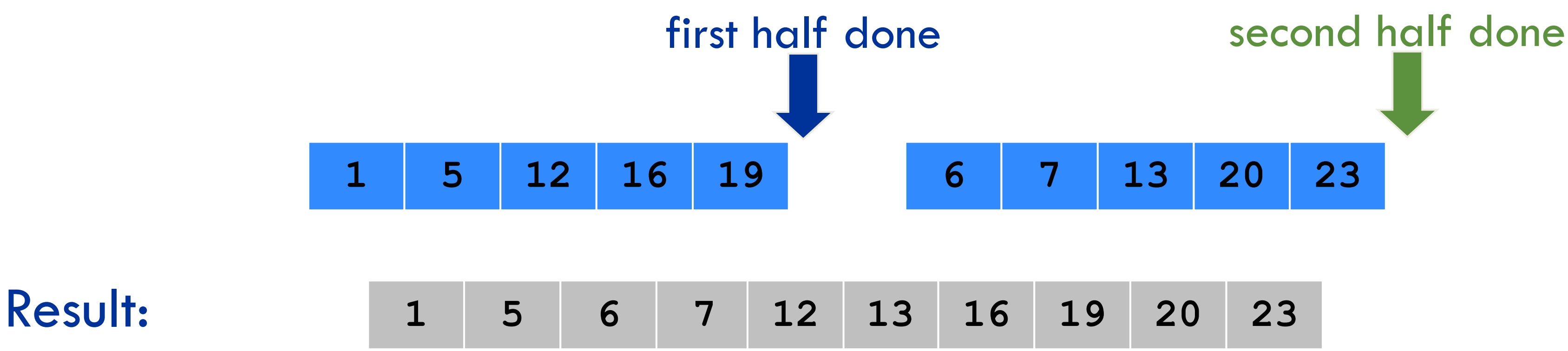
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge: Implementation

```
def merge(L, R):
   result ← array of length (|L| + |R|)
   l, r ← 0, 0
   while l + r < |result| do
      index ← l + r
      if r ≥ |R| or (l < |L| and L[l] < R[r]) then
         result[index] ← L[l]
         l ← l + 1
      else
         result[index] ← R[r]
         r ← r + 1
   return result
```

Merge: Correctness

Induction hypothesis:

- After the i-th iteration, our result contains the i smallest elements in sorted order

Base case:

- After 0 iterations, our result is empty, so it contains the 0 smallest elements in sorted order

Induction:

- Assume IH after iteration k, to prove it after iteration k+1
- Since both halves are sorted and we add the smallest element not already in result, result now contains the k+1 smallest elements
- Sorted order follows from the fact that both halves are sorted, thus adding the smallest element implies sorted order of result

Merge-Sort

1. Divide array into two halves.
2. Recur Recursively sort each half.
3. Conquer Merge two sorted halves to make a sorted whole.

1	12	5	16	19	7	23	6	13	20

1	12	5	16	19		7	23	6	13	20	divide
1	5	12	16	19		6	7	13	20	23	recur

1	5	6	7	12	13	16	19	20	23	conquer

Merge-Sort: Correctness

Induction hypothesis:

- Merge-Sort correctly sorts an array of size i

Base case:

- If our array has size 0 or 1, it's already sorted

Induction:

- Assume IH for all arrays up to size k, to prove it for array of size k+1
- Splitting the array in half gives us two array of size at most k, so by IH those are sorted correctly
- We proved that given two sorted arrays, Merge returns a correctly sorted array containing the elements of both arrays
- Hence, by running Merge on the two sorted halves, we sort the original array

Merge sort complexity analysis

Divide step (find middle and split) takes $O(n)$

Recur step (solve left and right subproblem) takes $2\ T(n/2)$

Conquer step (merge subarrays) takes $O(n)$

Now we can set up the recurrence for $T(n)$:

$$T(n) = \begin{cases} 2\ T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n \log n)$

Solving recurrences by unrolling

General strategy:

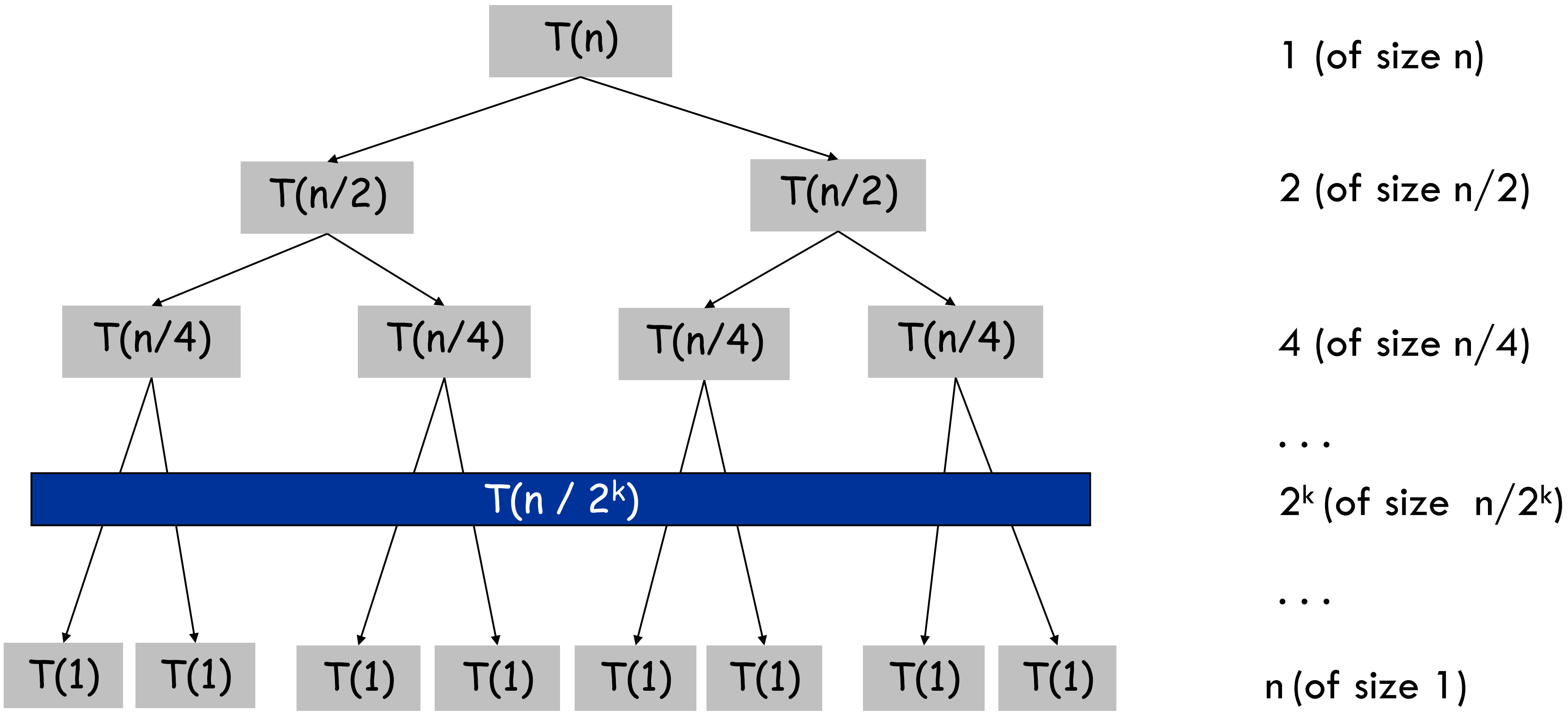
- Analyze first few levels
- Identify the pattern for a generic level
- Sum up over all levels

To verify the solution, we can substitute guess into the recurrence and prove it formally using induction if needed

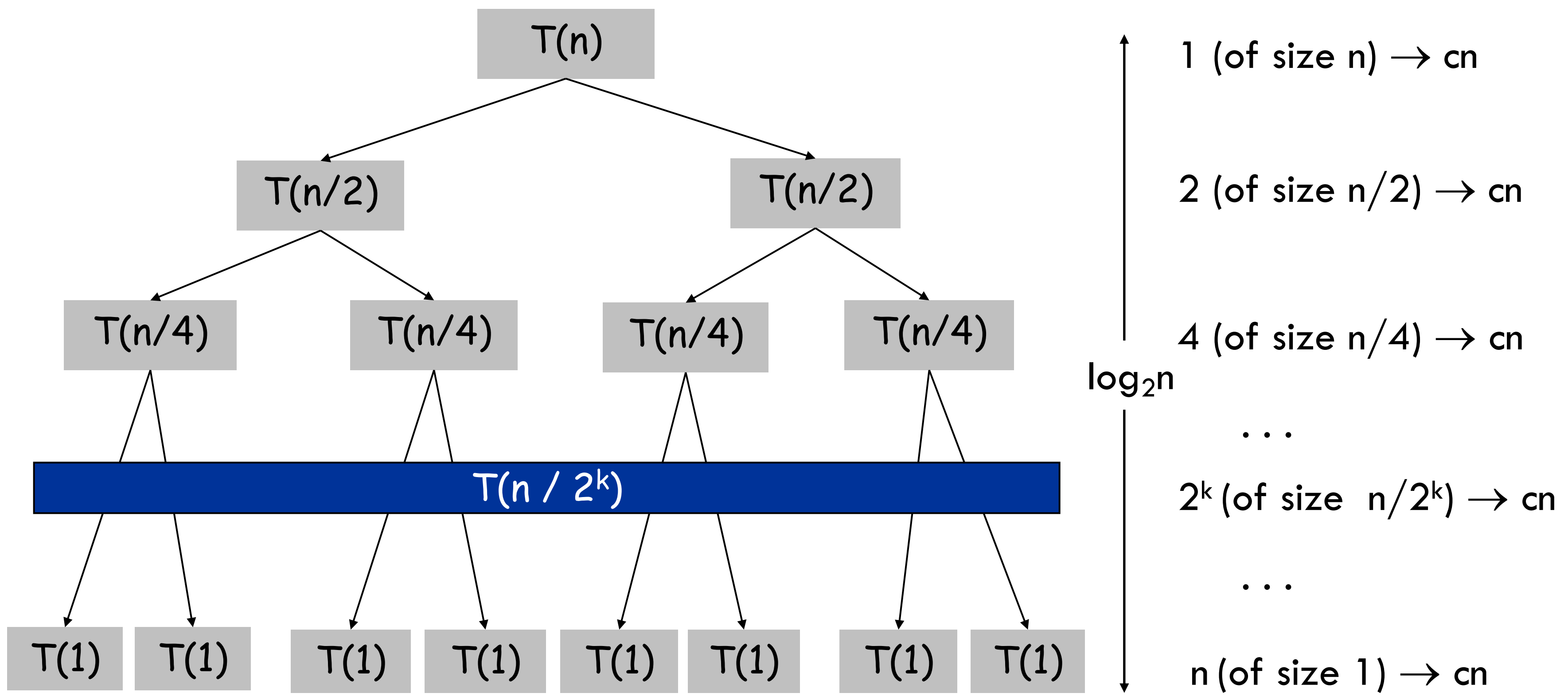
For Merge sort this method yields $T(n) = O(n \log n)$

There is a "Master theorem" (see textbook) that can handle most recurrences of interest, but unrolling is enough for our purposes

Proof by unrolling: T(n) = 2 T(n/2) + O(n)

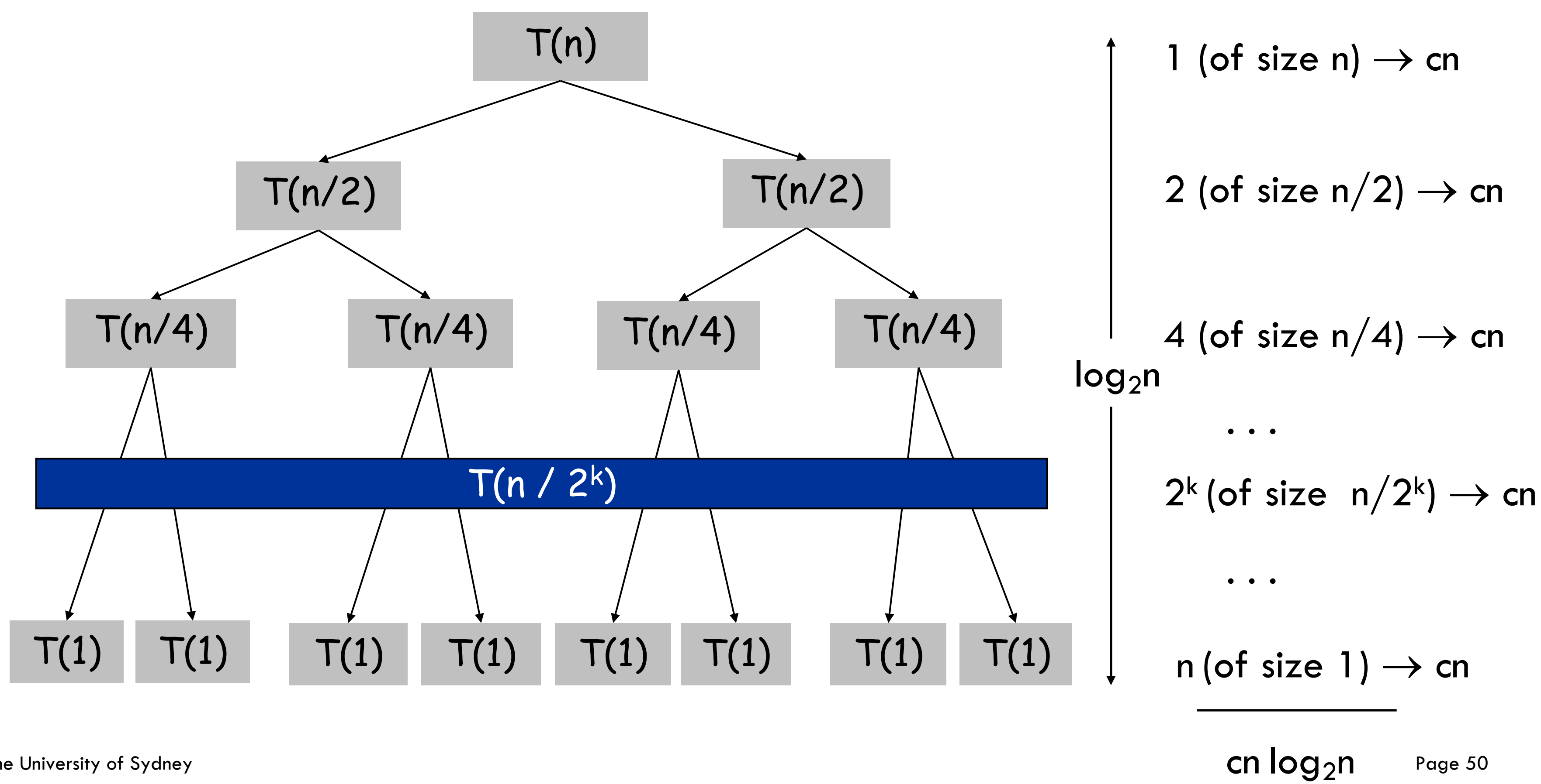


Proof by unrolling: T(n) = 2 T(n/2) + O(n)



Proof by unrolling: T(n) = 2 T(n/2) + O(n)

2^k c n/2^k = cn



Some recurrence formulas with solutions

Recurrence	Solution
$T(n) = 2\,T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = 2\,T(n/2) + O(\log n)$	$T(n) = O(n)$
$T(n) = 2\,T(n/2) + O(1)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(n)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$

What if n is not even?

Technically speaking we should be solving

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

If $n=2^k$, we would get the neat $T(n) = 2T(n/2) + O(n)$. But this is not always possible for more complicated recurrences.

For those cases, if we want to be formal, we have two options:

1. Solve the simpler recurrence to get a guess of a solution and then prove the solution by induction on the real recurrence
2. Define an auxiliary recurrence $S(n) = T(n + c)$ for some constant c. Show that $S(n) \leq 2\ S(n) + O(n)$

But you don't need to worry about all that in your proofs here.

Quick sort

It works better in practice than the merge sort altough having the same asymptotic time

universal at random

1. **Divide** Choose a random element from the list as the pivot

   Partition the elements into 3 lists:

   We can find the median element in an array and not random

   (i) less than, (ii) equal to and (iii) greater than the pivot

2. **Recur** Recursively sort the less than and greater than lists

3. **Conquer** Join the sorted 3 lists together

How to measure the performance of picking at random?
===
Measure if the pivot is decent: see if it falls in the middle 25-75% half. It's like flipping a coin.

We have three lists

pivot

1	7	6	16	19	7	13	5	13	19

concentration.

EV: O(nlogn)
deviating from EV will be exponential decrease

1	6	5		7	7		16	19	13	13	19

divide

1	5	6		7	7		13	13	16	19	19

recur

1	5	6	7	7	13	13	16	19	19

conquer

Quick sort complexity analysis

Divide step (pick pivot and split) takes $O(n)$

on the left and right L; R

Recur step (solve left and right subproblem) takes $T(n_L) + T(n_R)$

Conquer step (merge subarrays) takes $O(n)$

Only when the array is sorted

Now we can set up the recurrence for $T(n)$:

$$E[T(n)] = \begin{cases} E[T(n_L) + T(n_R)] + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

Expected running time

This solves to $E[T(n)] = O(n \log n)$ expected time

Quick sort detailed complexity analysis

Expected running time of quick sort is proportional to the expected number of comparisons in the algorithm.

T(n) = E[# comparisons of quick sort on array of size n]

comparison is n + comparisons for recursive calls

$$T(n) = n + \sum_{i=1}^{n} \frac{1}{n}(T(i-1) + T(n-i))$$

Quick sort detailed complexity analysis

T(n) = E[# comparisons of quick sort on array of size n]

$$T(n) = n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i)\right)$$

Note that every term if counted twice

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

Quick sort detailed complexity analysis

Multiply by n because reasons (it'll work out)

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

Take difference between n T(n) and (n-1) T(n-1)

$$\begin{aligned}
&n\,T(n) - (n-1)T(n-1) \\
&= n^2 + 2\sum_{i=1}^{n-1} T(i) - (n-1)^2 - 2\sum_{i=1}^{n-2} T(i) \\
&= 2n - 1 + 2T(n-1)
\end{aligned}$$

Quick sort detailed complexity analysis

$$n\,T(n) - (n-1)T(n-1) = 2n - 1 + 2T(n-1)$$

Getting back to n T(n)

$$n\,T(n) = 2n - 1 + (n+1)T(n-1)$$

Express T(n) in terms of T(n-1)

$$T(n) = \frac{2n-1}{n} + (n+1)\frac{T(n-1)}{n}$$

$$\frac{T(n)}{n+1} \leq \frac{2}{n+1} + \frac{T(n-1)}{n}$$

Quick sort detailed complexity analysis

$$\frac{T(n)}{n+1} \leq \frac{2}{n+1} + \frac{T(n-1)}{n}$$

Expanding the recursion

$$\frac{T(n)}{n+1} \leq \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots + 2$$

Do I spot a harmonic number?

$$T(n) \leq 2\,(n+1)\,H(n+1) = O(n \log n)$$

Harmonic number

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} = O(\log n)$$

Discretized integral of 1/i

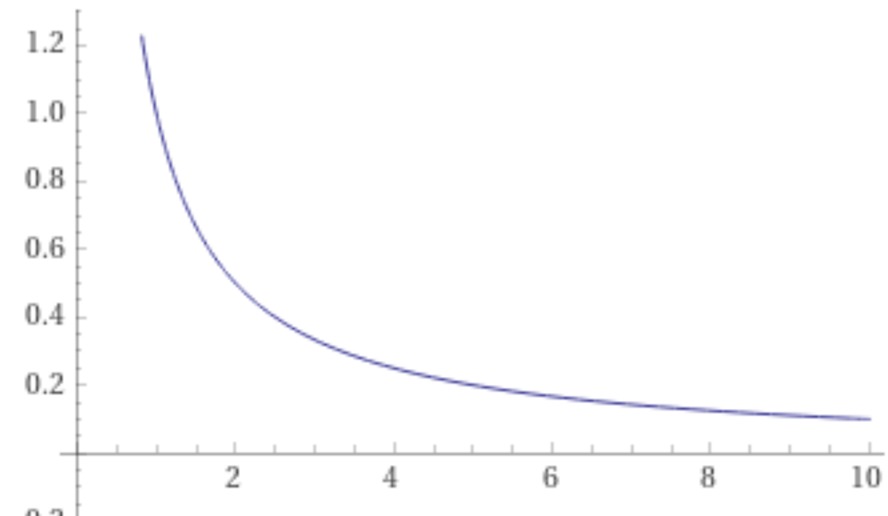


Image by Wolfram Alpha

Interlude: Comparison sorting lower bound

So far we've seen many sorting algorithms. Some run in $O(n^2)$ time while others run in $O(n \log n)$ time.

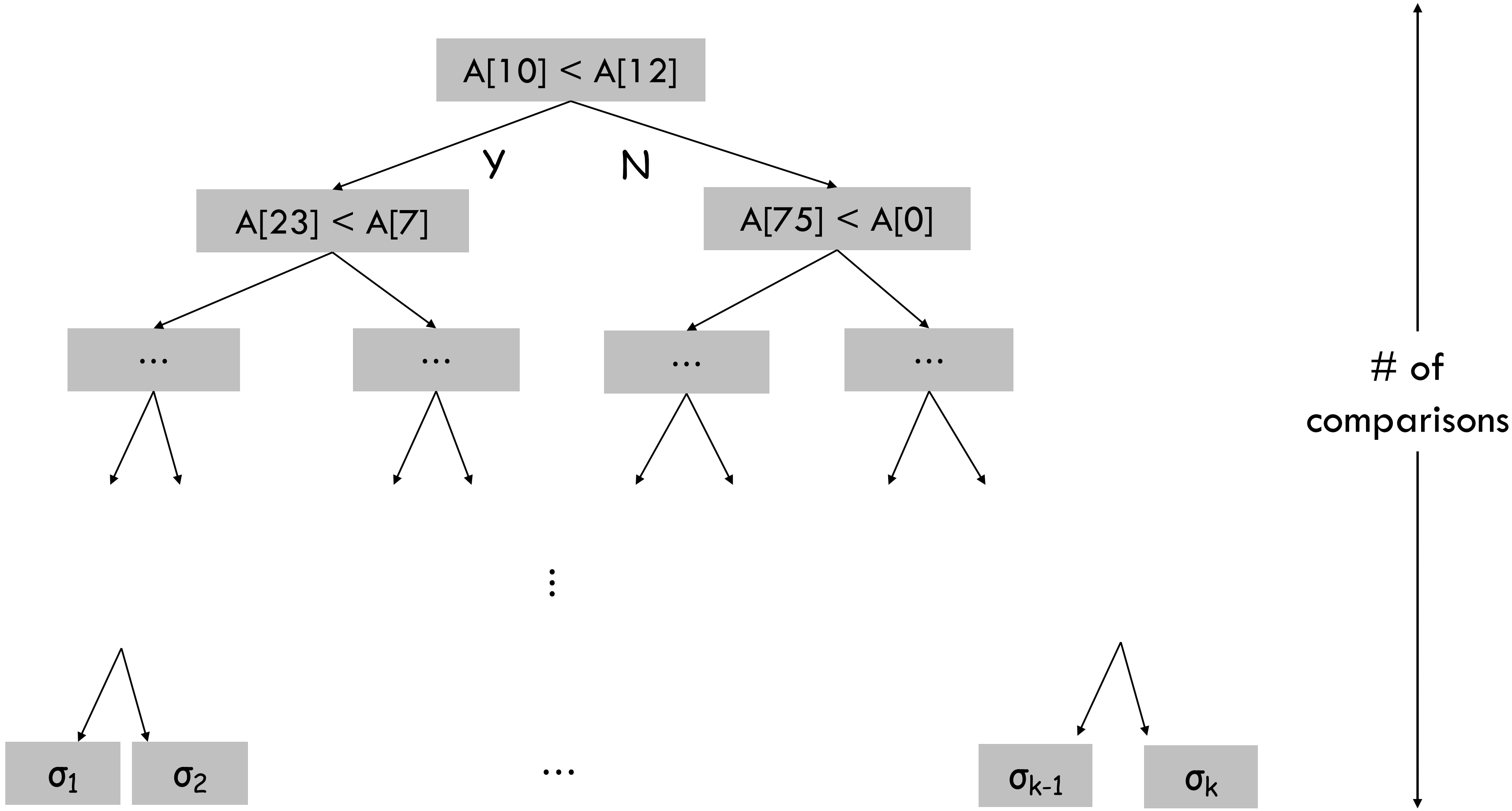
These algorithms work by performing pair-wise comparisons between elements of the sequence we are trying to sort

Such algorithms can be viewed as a decision tree where:

- each internal node compares two indices of the input array
- each external node corresponds to a permutation of $\{1,\ldots, n\}$

The height of the decision tree is a lower bound on the running time of the algorithm, since it only counts number of comparisons

Decision tree



The output of a leaf is A[σ(1)], A[σ(2)], … , A[σ(n)]

Interlude: Comparison sorting lower bound

Fact: Comparison-based sorting algorithms take $\Omega(n \log n)$ time

Proof:

The decision tree associated with a comparison-based sorting algorithm is binary and has at least n! external nodes. Thus the height is log n! which is $\Omega(n \log n)$

n! is the #permutation

$$\log n! = \log (n * (n-1) * \ldots * 1)$$ log n! is the #level, i.e., the height

$$= \log n + \log(n-1) + \ldots + \log 1$$

$$> n/2 * (\log n/2)$$ This is a very interesting inequality

$$= n/2 * ((\log n) - 1)$$

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