Further Mathematics and Algorithms

Lesson 16: *Sort Wisely*

Merge sort, quick sort and radix sort

Outline

1. **Merge Sort**
2. Quick Sort
3. Radix Sort

Merge Sort

- Merge sort is an example of sort performed in log-linear (i.e. $O(n\log(n)))$ time complexity
- It was invented in 1945 by John von Neumann
- It is an example of a divide-and-conquer strategy
  - ⋆ That is, the problem is divided into a number of parts recursively
  - ⋆ The full solution is obtained by recombining the parts

Algorithm

MergeSort($\boldsymbol{a}$)

{

if $n > 1$

copy $\boldsymbol{a}[1 : \lfloor n/2 \rfloor]$ **to** $\boldsymbol{b}$

copy $\boldsymbol{a}[\lfloor n/2 \rfloor + 1 : n]$ **to** $\boldsymbol{c}$

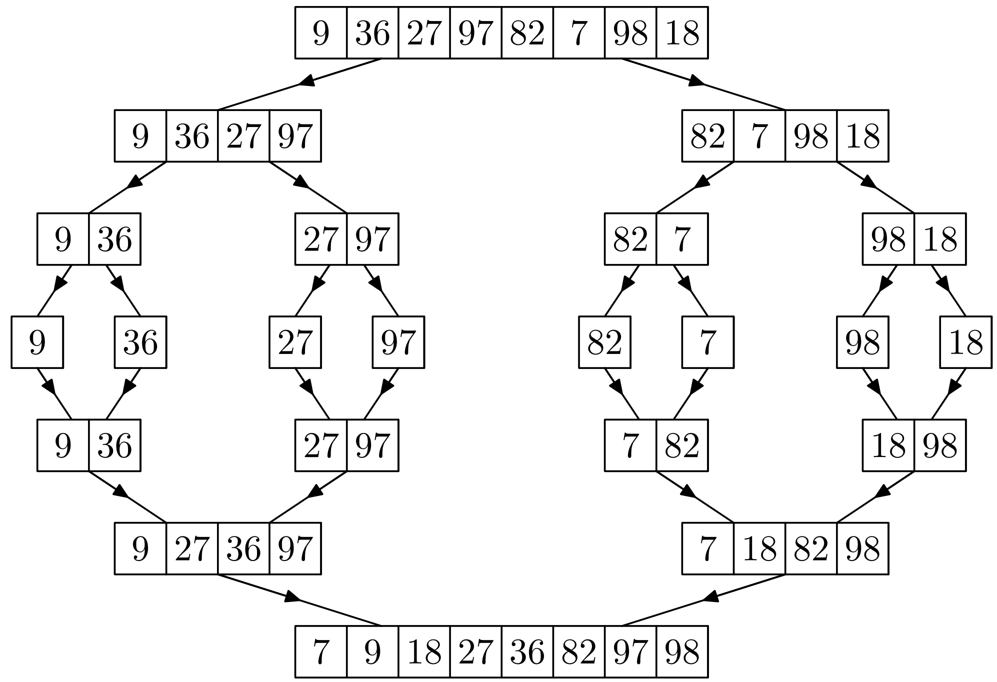
MergeSort($\boldsymbol{b}$)

MergeSort($\boldsymbol{c}$)

Merge($\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{a}$)

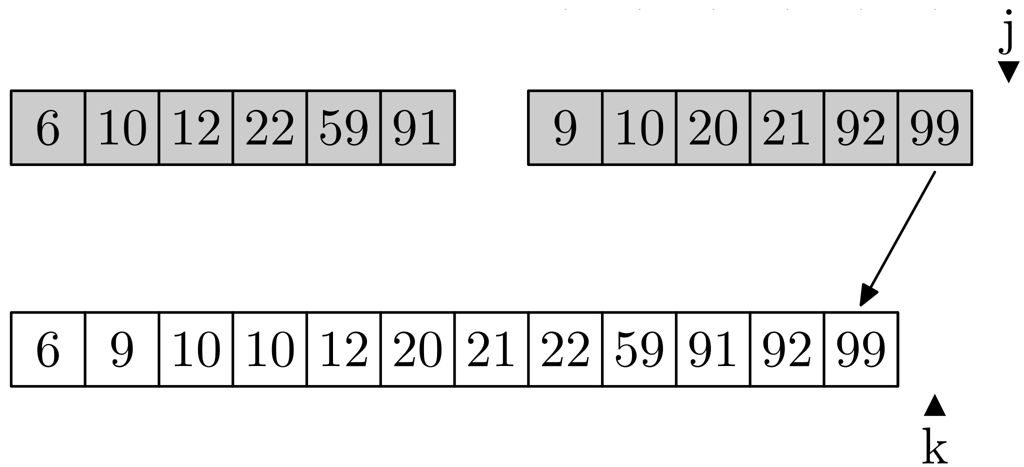
endif

}



Merge

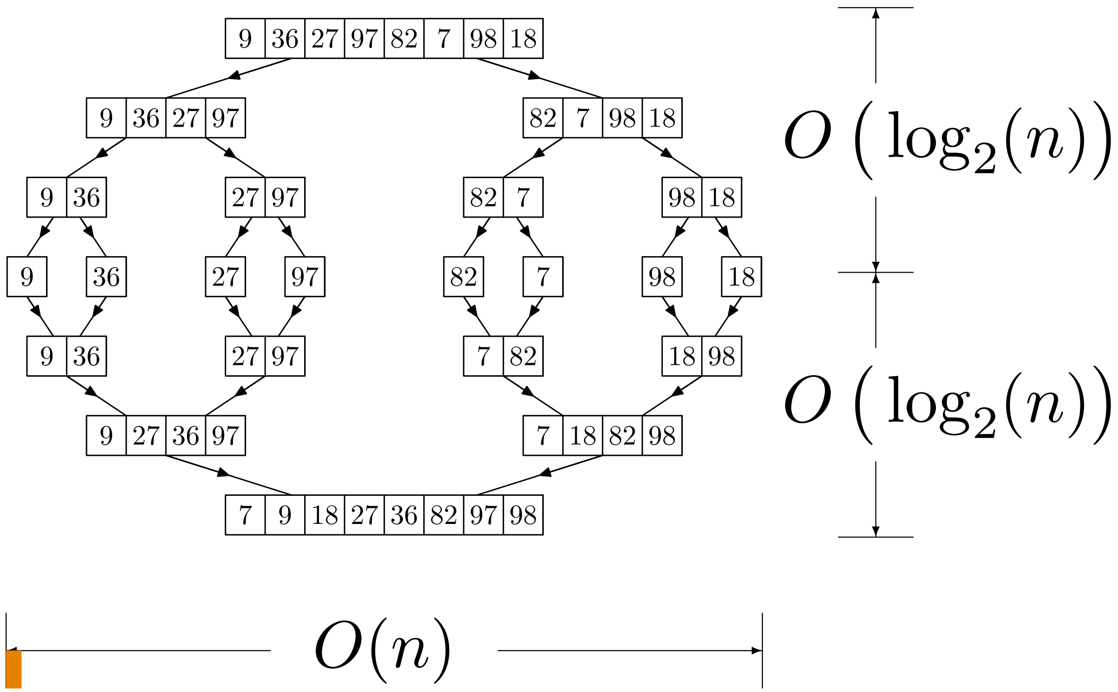
```
MERGE(b[1 : p], c[1 : q], a[1 : p + q])
{
  i←1
  j←1
  k←1
  while i ≤ p  and j ≤ q  do
    if b_i ≤ c_j
      a_k ←b_i
      i ←i+1
    else
      a_k ←c_j
      j ←j+1
    endif
    k ←k+1
  end
  if i=p
    copy c[j : q]  to a[k : p + q]
  else
    copy c[i : q]  to a[k : p + q]
}
```



Properties of Merge Sort

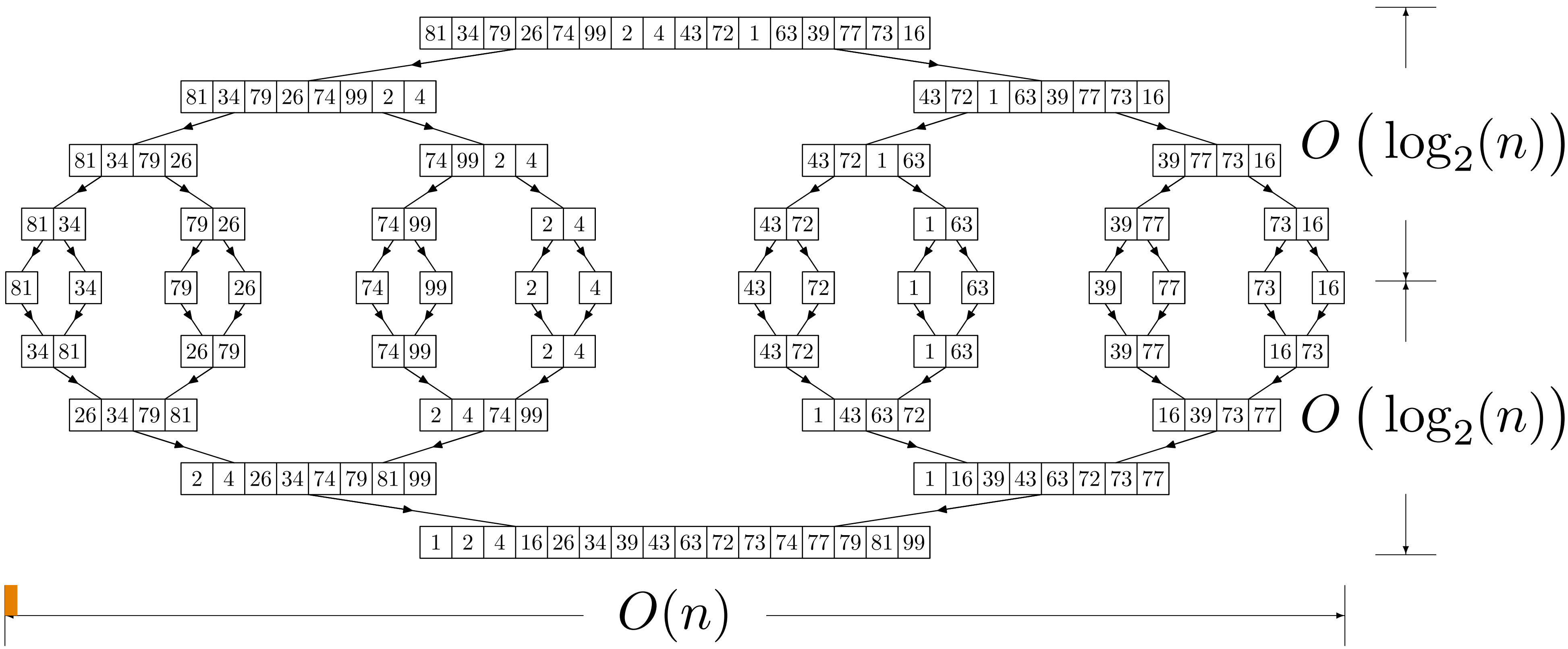
- Merge sort is stable provided we merge carefully (i.e. it preserves the order of two entries with the same value)
- Merge sort isn't in-place—we need an array of at most size n to do the merging
- Merging is quick. Given two arrays of size n the most number of comparisons we need to perform is $n - 1$

Time Complexity of Merge Sort



$$n = 8$$

Time Complexity of Merge Sort



$$n = 16$$

Time Complexity

- We again measure the complexity in the number of comparisons
- From the above argument $C(n) = O(n \times \log_2(n))$
- We can be a bit more formal

$$C(n) = 2C(\lfloor n/2 \rfloor) + C_{\text{merge}}(n) \qquad \text{for } n > 1$$
$$C(0) = 1$$

- But in the worst case $C_{\text{merge}}(n) = n - 1$
- Leads to $C_{\text{worst}}(n) = n \log_2(n) - n + 1$

General Time Complexity

- In general if we have a recursion formula

$$T(n) = aT(n/b) + f(n)$$

with $a \geq 1$, $b > 1$

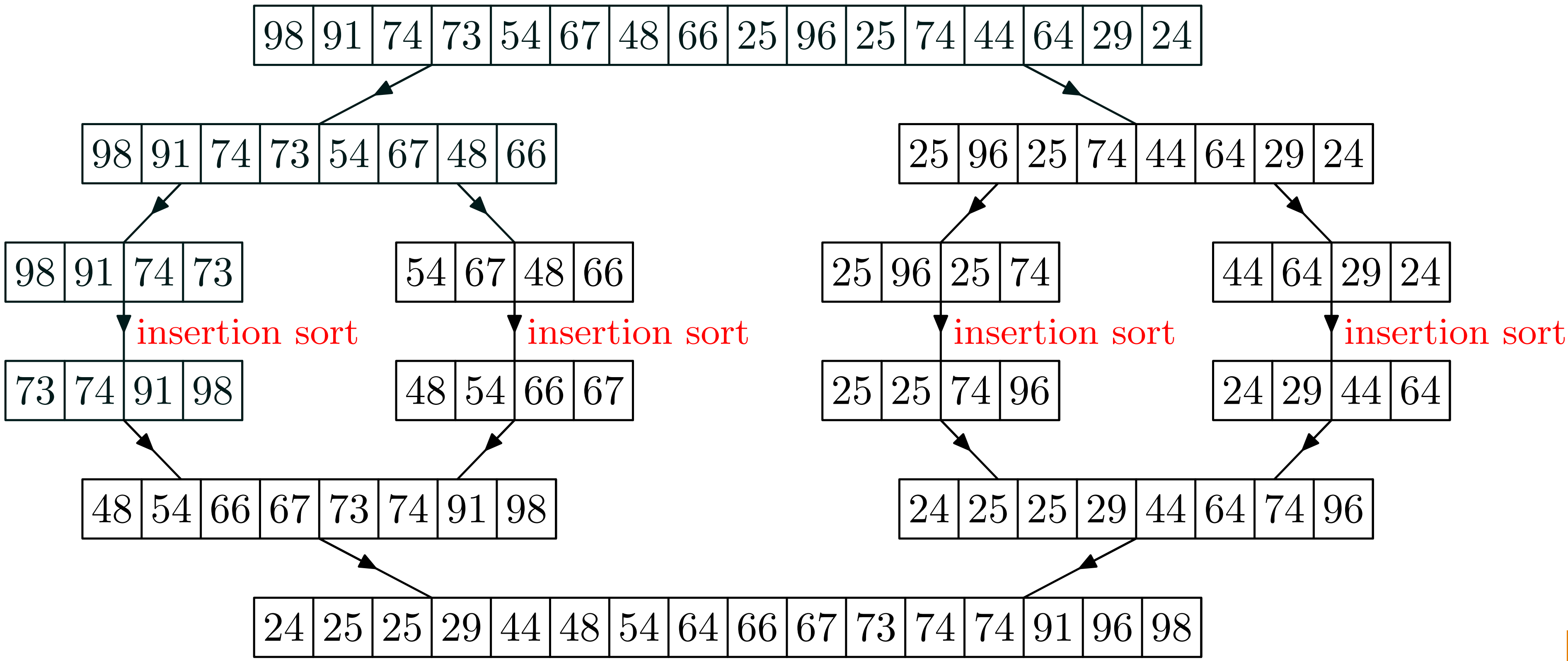
- If $f(n) \in \Theta(n^d)$ where $d \geq 0$ then

$$T(n) \in \begin{cases} \Theta\left(n^d\right) & \text{if } a < b^d \\ \Theta\left(n^d \log(n)\right) & \text{if } a = b^d \\ \Theta\left(n^{\log_d(a)}\right) & \text{if } a > b^d \end{cases}$$

- Analogous results hold for the family O and Ω

Mixing Sort

- For very short sequences it is faster to use insertion sort than to pay the overhead of function calls



Outline

1. Merge Sort

2. **Quick Sort**

3. Radix Sort

Quicksort

- The most commonly used fast sorting algorithm is **quicksort**
- It was invented by the British computer scientist by C. A. R. Hoare in 1962
- It again uses the divide-and-conquer strategy
- It can be performed in-place, but it is **not** stable
- It works by splitting an array into two depending on whether the elements are less than or greater than a **pivot** value
- This is done recursively until the full array is sorted

Partition

- We need to partition the array around the pivot p such that

all elements < p	all elements >= p

```
PARTITION(a, p, left, right)
{
  i ←left
  j ←right
  repeat {
    while a_i < p
      i++
    while a_j ≥ p
      j--
    if i ≥ j
      break
    SWAP(a_i, a_j)
  }
}
```

pivot $=$ 52

33	49	11	29	40	6	47	10	73	60	87	96	52	94	57	85

j

Optimising Partitioning

- There are different ways of performing the partitioning
- We want to minimise the time taken on the inner loop
- This means we want to perform as few checks as possible
- One method of doing this is to place *sentinels* at the ends of the array
- We can also reduce work by placing the partition in its correct position

all elements <= p	p	all elements >= p

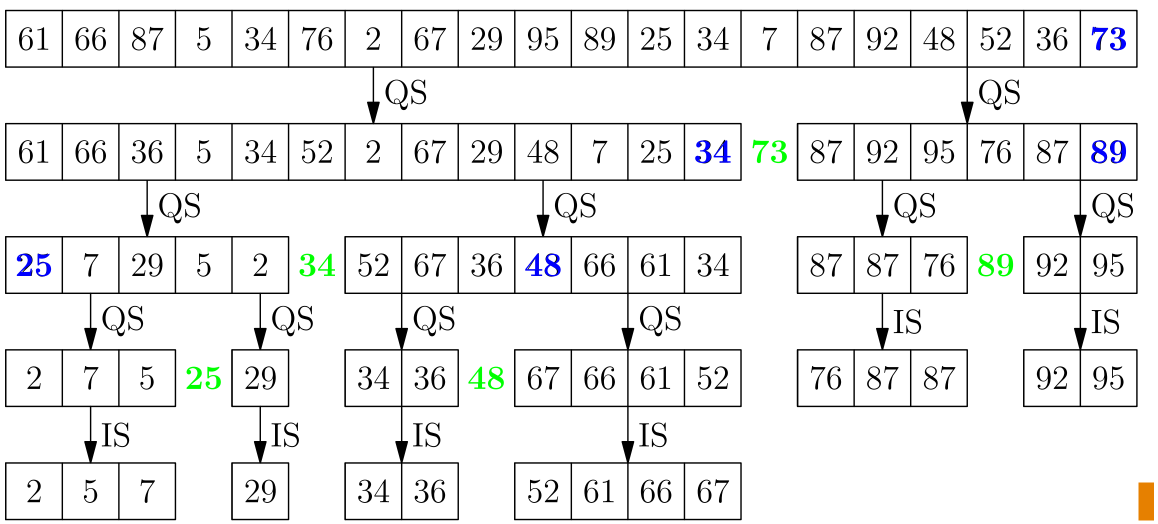
Choosing the Pivot

- There are different strategies to choosing the pivot
- Choose the first element in the array
- Choose the median of the first, middle and last element of the array
- This increases the likelihood of the pivot being close to the median of the whole array
- For large arrays (above 40) the median of 3 medians is often used

Quicksort

We recursively partition the array until each partition is small enough to sort using insertion sort

```
QUICKSORT(a, left, right) {
  if (right-left < threshold)
    INSERTIONSORT(a, left, right)
  else
    pivot = CHOOSEPIVOT(a, left, right)
    part = PARTITION(a, pivot, left, right)
    QUICKSORT(a, left, part-1)
    QUICKSORT(a, part+1, right)
  endif
}
```



Time Complexity

- Partitioning an array of size n takes $\Theta(n)$ operations
- If we split the array in half then number of partitions we need to do is $\lceil \log_2(n) \rceil$
- This is the best case thus quicksort is $\Omega\left(n \log(n)\right)$
- If the pivot is the minimum element of the array then we have to partition $n-1$ times
- This is the worst case so quicksort is $O\left(n^2\right)$
- This worst case will happen if the array is already sorted and we choose the pivot to be the first element in the array!

QuickSort

```
0  quickSort(a, [illegible], [illegible]){
1    if([illegible]){
2      p = choosePivot(a, [illegible], [illegible])
3      i = partition(a, [illegible], [illegible], [illegible])
4      quickSort(a, [illegible], [illegible])
5      quickSort(a, [illegible], [illegible])
6    } else
7      insertionSort(a, [illegible], [illegible])
8    return
9  }
```

```
PC = [illegible]
l = [illegible]
h = [illegible]
p = [illegible]
i = [illegible]
```

pc	l	h	p	i
7	[illegible]	[illegible]	#	#
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	0	19	[illegible]	[illegible]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	5	7	25	29	34	34	36	48	52	61	66	67	73	76	87	87	89	92	95
low		high		high / low		low	high		low			high		low		high		low	high

Sort in Practice

- The STL in C++ offers three sorts
  - ⋆ `sort()` implemented using quicksort
  - ⋆ `stable_sort()` implemented using mergesort
  - ⋆ `partial_sort()` implemented using heapsort
- Java uses
  - ⋆ Quicksort to sort arrays of primitive types
  - ⋆ Mergesort to sort Collections of objects
- Quicksort is typically fastest but has worst case quadratic time complexity

Selection

- A related problem to sorting is selection
- That is we want to select the k^{th} largest element
- We could do this by first sorting the array
- A full sort is not however necessary—we can use a modified quicksort where we only continue to sort the part of the array we are interested in
- This leads to a $\Theta(n\log(n))$ algorithm which is considerably faster then sorting

Outline

1. Merge Sort

2. Quick Sort

3. **Radix Sort**

Radix Sort

- Can we get a sort algorithm to run faster than $O\left(n\log(n)\right)$?
- Our proof that this was optimal assumed we were performing binary decisions (is a_i less than a_j?)
- If we don't perform pairwise comparisons then the proof doesn't apply
- Radix sort is the classic example of a sort algorithm that doesn't use pairwise comparisons

Sorting Into Buckets

- The idea behind radix sort is to sort the elements of an array into some number of buckets
- This is done successively until the whole array is sorted
- Consider sorting integers in decimals (base 10 or radix 10)
- We can successively sort on the digits
- The sort finishes when we have got through all the digits

Radix Sort in Action

11	
13	
26	
29	
37	
43	
51	
51	
52	
79	

0	null
1	null
2	null
3	null
4	null
5	null
6	null
7	null
8	null
9	null

Time Complexity of Radix Sort

- We need not use base 10 we could use base r (the radix)
- If the maximum number to be sorted is N then the number of iterations of radix sort is $\log_r(N)$
- Each sort involves n operations
- Thus the total number of operations is $O\left(n\lceil\log_r(N)\rceil\right)$
- Since N does not depend on n we can write this as $O\left(n\right)$

Bucket Sort

- A closely related sort is bucket sort where we divide up the inputs into buckets based on the most significant figure
- We then sort the buckets on less significant figures
- Quicksort is a bucket sort with two buckets, but where we choose a pivot to determine which bucket to use

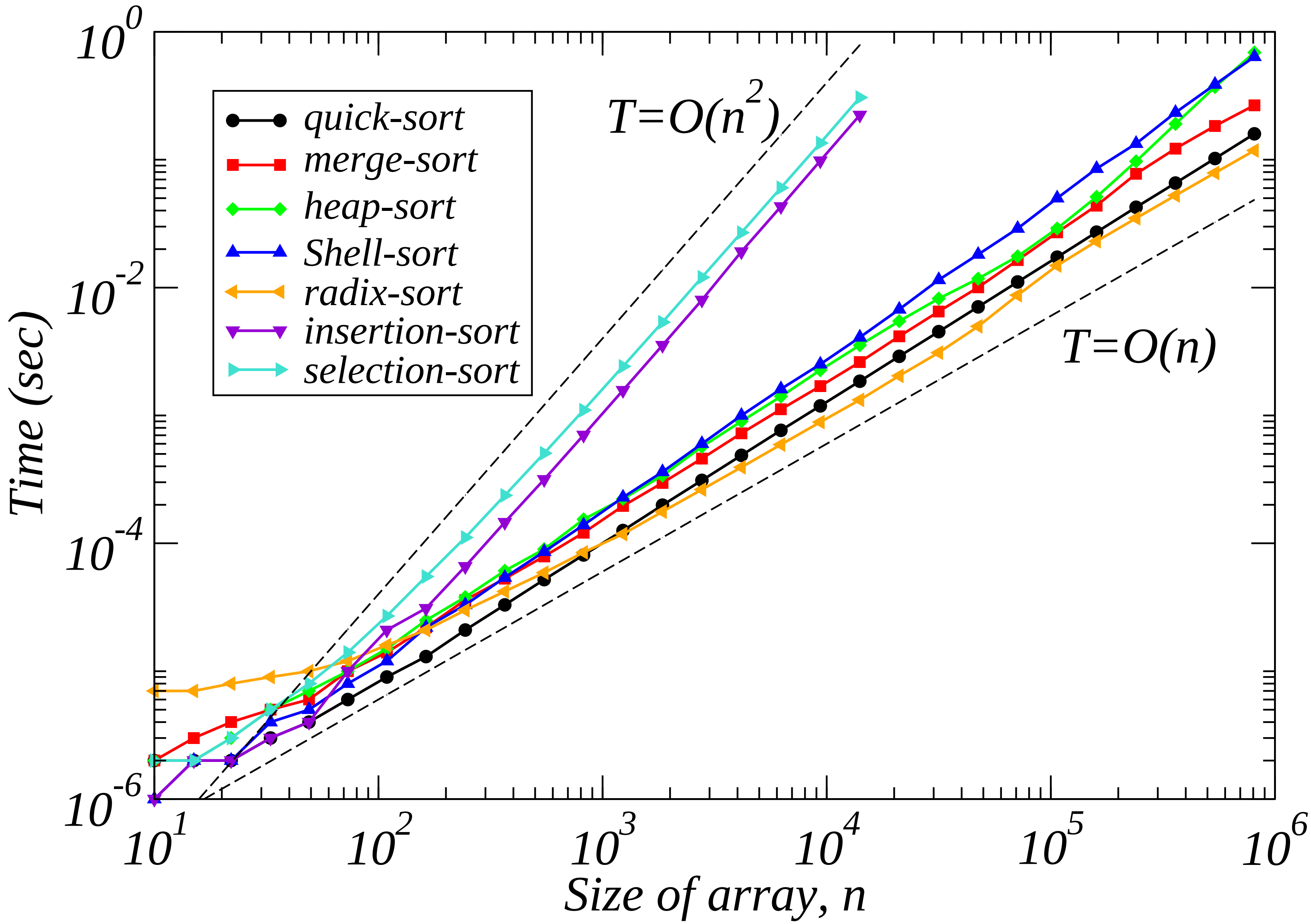
Minimum Time for Sort

- Can we do better?
- In any sort we need to examine all possible elements in the array
- If there is an element that isn't examined then we don't know where to put it
- Thus the lower bound on any sort algorithm is $\Omega(n)$

Practical Sort

- In practice, radix sort or bucket sort are rarely used
- The overhead of maintaining the buckets make them less efficient than they might appear
- Radix sort is harder to generalise to other data types than comparison based sorts
- In practice quick sort and merge sort are usually preferred
- Having said that there are some very neat implementations of radix sort

Comparison of Sort Algorithms



Lessons

- Sort is important—it is one of the commonest high level operations
- Merge sort and quick sort are the most commonly used sort
- There are sorts that have a better time complexity that quicksort
- In practice it is difficult to beat quicksort