Algorithm Efficiency and Sorting

Measuring the Efficiency of Algorithms

- Analysis of algorithms
  - contrasts the efficiency of different methods of solution
- A comparison of algorithms
  - Should focus on significant differences in efficiency
  - Should not consider reductions in computing costs due to clever coding tricks

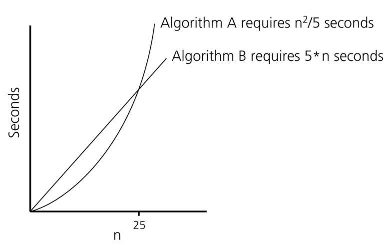
The Execution Time of Algorithms

- Counting an algorithm's operations is a way to access its efficiency
  - An algorithm's execution time is related to the number of operations it requires

Algorithm Growth Rates

- An algorithm's time requirements can be measured as a function of the input size
- Algorithm efficiency is typically a concern for large data sets only

Algorithm Growth Rates

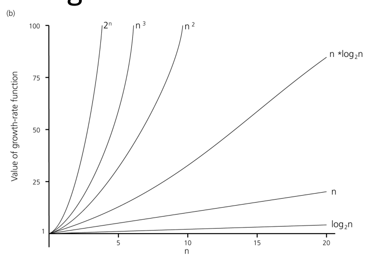


Time requirements as a function of the problem size *n*

Order-of-Magnitude Analysis and Big O Notation

- Definition of the order of an algorithm
  - Algorithm A is order f(n) – denoted O(f(n)) – if constants k and n_0 exist such that A requires no more than k * f(n) time units to solve a problem of size $n \geq n_0$
- Big O notation
  - A notation that uses the capital letter O to specify an algorithm's order of growth
  - Example: O(f(n))

Order-of-Magnitude Analysis and Big O Notation



A comparison of growth-rate functions: b) in graphical form

Order-of-Magnitude Analysis and Big O Notation

- Order of growth of some common functions
  $O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < O(2^n)$
- Properties of growth-rate functions
  - You can ignore low-order terms
  - You can ignore a multiplicative constant in the high-order term
  - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

Order-of-Magnitude Analysis and Big O Notation

- Worst-case analyses
  - An algorithm can require different times to solve different problems of the same size
    - Worst-case analysis
      - A determination of the maximum amount of time that an algorithm requires to solve problems of size n

The Efficiency of Searching Algorithms

- Sequential search
  - Strategy
    - Look at each item in the data collection in turn, beginning with the first one
    - Stop when
      - You find the desired item
      - You reach the end of the data collection

The Efficiency of Searching Algorithms

- Sequential search
  - Efficiency
    - Worst case: O(n)
    - Average case: O(n)
    - Best case: O(1)

The Efficiency of Searching Algorithms

- Binary search
  - Strategy
    - To search a sorted array for a particular item
      - Repeatedly divide the array in half
      - Determine which half the item must be in, if it is indeed present, and discard the other half
  - Efficiency
    - Worst case: $O(\log_2 n)$

Sorting Algorithms and Their Efficiency

- Sorting
  - A process that organizes a collection of data into either ascending or descending order
- Categories of sorting algorithms
  - An internal sort
    - Requires that the collection of data fit entirely in the computer's main memory. Called in-place if it uses space proportional to data size
  - An external sort
    - The collection of data will not fit in the computer's main memory all at once but must reside in secondary storage

Sorting Algorithms and Their Efficiency

- Data items to be sorted can be
  - Integers
  - Character strings
  - Objects
- Sort key
  - The part of a record that determines the sorted order of the entire record within a collection of records

Selection Sort

- Selection sort
  - Strategy
    - Select the largest item and put it in its correct place
    - Select the next largest item and put it in its correct place, etc.

Shaded elements are selected;
boldface elements are in order.

Initial array:	29	10	14	37	13
After 1st swap:	29	10	14	13	**37**
After 2nd swap:	13	10	14	**29**	**37**
After 3rd swap:	13	10	**14**	**29**	**37**
After 4th swap:	**10**	**13**	**14**	**29**	**37**

A selection sort of an array of five integers

Selection Sort

- Analysis
  - Selection sort is $O(n^2)$
- Advantage of selection sort
  - The running time does not depend on the initial arrangement of the data (worst case running time is same as best case running time on all data sets)
- Disadvantage of selection sort
  - It is only appropriate for small n

Bubble Sort

- Bubble sort
  - Strategy
    - Compare adjacent elements and exchange them if they are out of order
      - Comparing the first two elements, the second and third elements, and so on, will move the largest elements to the end of the array
      - Repeating this process will eventually sort the array into ascending order

Bubble Sort

(a) Pass 1

Initial array:	29	10	14	37	13
	10	29	14	37	13
	10	14	29	37	13
	10	14	29	37	13
	10	14	29	13	**37**

(b) Pass 2

10	14	29	13	**37**
10	14	29	13	**37**
10	14	29	13	**37**
10	14	13	**29**	**37**

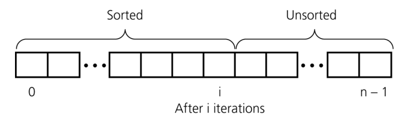
The first two passes of a bubble sort of an array of five integers: a) pass 1; b) pass 2

Bubble Sort

- Analysis
  - Worst case: $O(n^2)$
  - Best case: $O(n)$

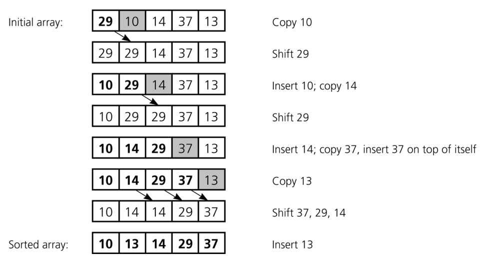
Insertion Sort

- Insertion sort
  - Strategy
    - Partition the array into two regions: sorted and unsorted
    - Take each item from the unsorted region and insert it into its correct order in the sorted region



An insertion sort partitions the array into two regions

Insertion Sort



An insertion sort of an array of five integers.

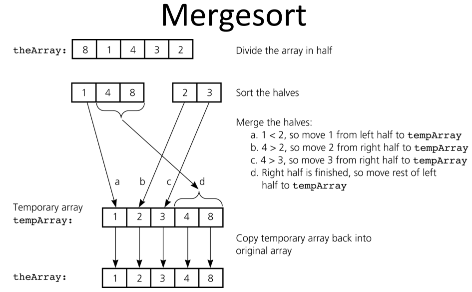
Insertion Sort

- Analysis
  - Worst case: $O(n^2)$
  - For small arrays
    - Insertion sort is appropriate due to its simplicity
  - For large arrays
    - Insertion sort is prohibitively inefficient

Mergesort

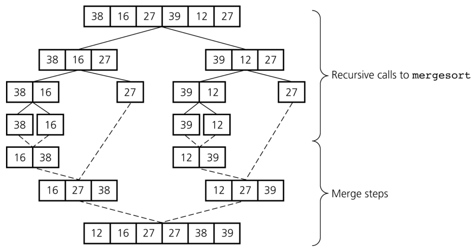
- Important divide-and-conquer sorting algorithms
  - Mergesort
  - Quicksort
- Mergesort
  - A recursive sorting algorithm
  - Gives the same performance, regardless of the initial order of the array items
  - Strategy
    - Divide an array into halves
    - Sort each half
    - Merge the sorted halves into one sorted array

Mergesort



A mergesort with an auxiliary temporary array

Mergesort



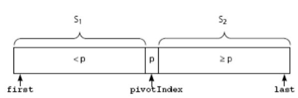
A mergesort of an array of six integers

Mergesort

- Analysis
  - Worst case: O(n $\log_2 n$)
  - Average case: O(n $\log_2 n$)
  - Advantage
    - It is an extremely efficient algorithm with respect to time
  - Drawback
    - It requires a second array as large as the original array

Quicksort

- Quicksort
  - A divide-and-conquer algorithm
  - Strategy
    - Partition an array into items that are less than the pivot and those that are greater than or equal to the pivot
    - Sort the left section
    - Sort the right section

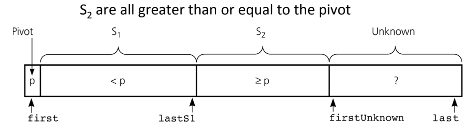


A partition about a pivot

Quicksort

- Using an invariant to develop a partition algorithm
  - Invariant for the partition algorithm

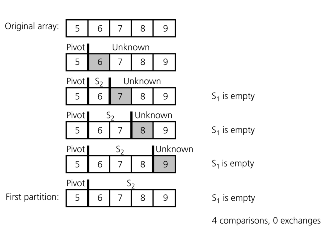
    The items in region S_1 are all less than the pivot, and those in S_2 are all greater than or equal to the pivot



Invariant for the partition algorithm

Quicksort

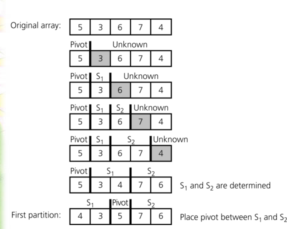
- Analysis
  - Worst case
    - `quicksort` is $O(n^2)$ when the array is already sorted and the smallest item is chosen as the pivot



A worst-case partitioning with ***quicksort***

Quicksort

- Analysis
  - Average case
    - `quicksort` is O(n * $\log_2 n$) when S_1 and S_2 contain the same – or nearly the same – number of items arranged at random



A average-case partitioning with ***quicksort***

Quicksort

- Analysis
  - `quicksort` is usually extremely fast in practice
  - Even if the worst case occurs, `quicksort`'s performance is acceptable for moderately large arrays

Radix Sort

- Radix sort
  - Treats each data element as a character string
  - Strategy
    - Repeatedly organize the data into groups according to the i^{th} character in each element
- Analysis
  - Radix sort is O(n)

Radix Sort

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(156**0**, 215**0**) (106**1**) (022**2**) (012**3**, 028**3**) (215**4**, 000**4**)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(00**0**4) (02**2**2, 01**2**3) (21**5**0, 21**5**4) (15**6**0, 10**6**1) (02**8**3)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0**0**04, 1**0**61) (0**1**23, 2**1**50, 2**1**54) (0**2**22, 0**2**83) (1**5**60)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(**0**004, **0**123, **0**222, **0**283) (**1**061, **1**560) (**2**150, **2**154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

A radix sort of eight integers

A Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	n * log n	n * log n
Quicksort	n^2	n * log n
Radix sort	n	n
Treesort	n^2	n * log n
Heapsort	n * log n	n * log n

Approximate growth rates of time required for eight sorting algorithms

Summary

- Worst-case and average-case analyses
  - Worst-case analysis considers the maximum amount of work an algorithm requires on a problem of a given size
  - Average-case analysis considers the expected amount of work an algorithm requires on a problem of a given size
- Order-of-magnitude (aka asymptotic) analysis can be used to choose an implementation for an abstract data type
- Selection sort, bubble sort, and insertion sort are all $O(n^2)$ algorithms
- Quicksort and mergesort are two very efficient sorting algorithms ($O(n\log_2 n)$)