Algorithm Efficiency & Sorting

- Algorithm efficiency
- Big-O notation
- Searching algorithms
- Sorting algorithms

Overview

- Writing programs to solve problem consists of a large number of decisions
  - how to represent aspects of the problem for solution
  - which of several approaches to a given solution component to use
- If several algorithms are available for solving a given problem, the developer must choose among them
- If several ADTs can be used to represent a given set of problem data
  - which ADT should be used?
  - how will ADT choice affect algorithm choice?

Overview – 2

- If a given ADT (i.e. stack or queue) is attractive as part of a solution
- How will the ADT implemention affect the program's:
  - correctness and performance?
- Several goals must be balanced by a developer in producing a solution to a problem
  - correctness, clarity, and efficient use of computer resources to produce the best performance
- How is solution performance best measured?
  - *time* and *space*

Overview – 3

- The order of importance is, generally,
  - correctness
  - efficiency
  - clarity
- Clarity of expression is qualitative and somewhat dependent on perception by the reader
  - developer salary costs dominate many software projects
  - time efficiency of understanding code written by others can thus have a significant monetary implication
- Focus of this chapter is *execution efficiency*
  - mostly, run-time (some times, memory space)

Measuring Algorithmic Efficiency

- Analysis of algorithms
  - provides tools for contrasting the efficiency of different methods of solution
- Comparison of algorithms
  - should focus on *significant* differences in efficiency
  - should not consider reductions in computing costs due to clever coding tricks
- Difficult to compare programs instead of algorithms
  - how are the algorithms coded?
  - what computer should you use?
  - what data should the programs use?

Analyzing Algorithmic Cost

- Viewed abstractly, an algorithm is a sequence of steps
  - Algorithm A { S1; S2; .... Sm1; Sm }
- The total cost of the algorithm will thus, obviously, be the total cost of the algorithm's *m* steps
  - assume we have a function giving cost of each statement

    Cost (S_i) = execution cost of S_i, for-all i, $1 \leq i \leq m$
- Total cost of the algorithm's m steps would thus be:

  $Cost\ (A) = \sum_{i=1}^{m} Cost\ (Si)$

Analyzing Algorithmic Cost – 2

- However, an algorithm can be applied to a wide variety of problems and data sizes
  - so we want a cost function for the algorithm A that takes the data set size *n* into account

    $Cost\ (A, n) = \sum_{1}^{n}\ (\sum_{1}^{m}(Cost\ (S_i)))$
- Several factors complicate things
  - conditional statements: cost of evaluating condition and branch taken
  - loops: cost is sum of each of its iterations
  - recursion: may require solving a recurrence equation

Analyzing Algorithmic Cost – 3

- Do not attempt to accumulate a precise prediction for program execution time, because
  - far too many complicating factors: compiler instructions output, variation with specific data sets, target hardware speed
- Provide an approximation, an *order of magnitude* estimate, that permits fair comparison of one algorithm's behavior against that of another

Analyzing Algorithmic Cost – 4

- Various behavior bounds are of interest
  - best case, average case, worst case
- Worst-case analysis
  - A determination of the maximum amount of time that an algorithm requires to solve problems of size n
- Average-case analysis
  - A determination of the average amount of time that an algorithm requires to solve problems of size n
- Best-case analysis
  - A determination of the minimum amount of time that an algorithm requires to solve problems of size n

Analyzing Algorithmic Cost – 5

- Complexity measures can be calculated in terms of
  - T(n): time complexity and S(n): space complexity
- Basic model of computation used
  - sequential computer (one statement at a time)
  - all data require same amount of storage in memory
  - each datum in memory can be accessed in constant time
  - each basic operation can be executed in constant time
- Note that all of these assumptions are incorrect!
  - good for this purpose
- Calculations we want are order of magnitude

Example – Linked List Traversal

- Assumptions
  C_1 = cost of assign.
  C_2 = cost of compare
  C_3 = cost of write

```
Node *cur = head;       // assignment op
  while (cur != NULL) // comparisons op
  cout << cur→item
       << endl;       // write op
  cur→next;           // assignment op
}
```

- Consider the number of operations for n items

$$T(n) = (n+1)C_1 + (n+1)C_2 + nC_3$$
$$= (C_1+C_2+C_3)n + (C_1+C_2) = K_1n + K_2$$

- Says, algorithm is of linear complexity
  - work done grows linearly with n but also involves constants

Example – Sequential Search

- Number of comparisons
  $T_B(n) = 1$
  $T_w(n) = n$
  $T_A(n) = (n+1)/2$
- In general, what developers worry about the most is that this is O(n) algorithm
  - more precise analysis is nice but rarely influences algorithmic decision

```
Seq_Search(A: array, key: integer);
    i = 1;
    while i ≤ n and A[i] ≠ key do
        i = i + 1
    endwhile;
    if i ≤ n
        then return(i)
        else return(0)
    endif;
end Sequential_Search;
```

Bounding Functions

- To provide a guaranteed bound on how much work is involved in applying an algorithm **A** to **n** items
  - we find a bounding function f(n) such that

    $$T(n) \leq f(n), \forall\, n$$
- It is often easier to satisfy a less stringent constraint by finding an elementary function f(n) such that

  $$T(n) \leq k\ *\ f(n), for\ sufficiently\ large\ n$$
- This is denoted by the asymptotic **big-O** notation
- Algorithm A is O(n) says
  - that complexity of A is no worse than k*n as n grows sufficiently large

Asymptotic Upper Bound

- Defn: A function f is positive if $f(n) > 0, \forall\, n > 0$
- Defn: Given a positive function f(n), then

  $$f(n) = O\big(g(n)\big)$$

  iff there exist constants k > 0 and n_0 > 0 such that

  $$f(n) \leq k * g(n), \forall\, n > n_0$$
- Thus, g(n) is an asymptotic bounding function for the work done by the algorithm
- k and n_0 can be *any* constants
  - can lead to unsatisfactory conclusions if they are very large and a developer's data set is relatively small

Asymptotic Upper Bound – 2

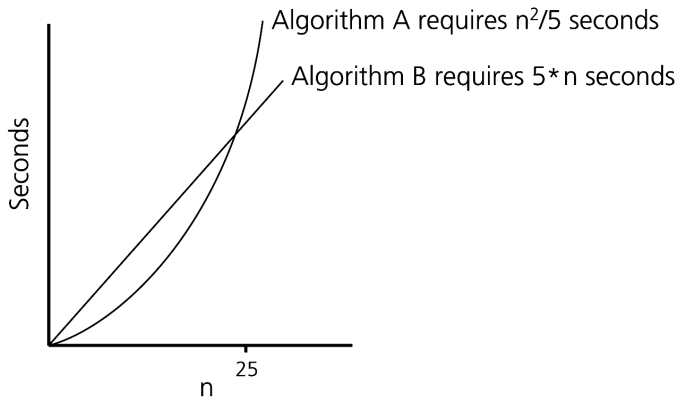
- Example: show that: $2n^2 - 3n + 10 = O(n^2)$
- Observe that

  $$2n^2 - 3n + 10 \leq 2n^2 + 10, n > 1$$
  $$2n^2 - 3n + 10 \leq 2n^2 + 10, n^2\ n > 1$$
  $$2n^2 - 3n + 10 \leq 12n^2, n > 1$$
- Thus, expression is $O(n^2)$ for k = 12 and n_0 > 1 (also k = 3 and n_0 > 1, BTW)
  - algorithm efficiency is typically a concern for large problems only
- Then, O(f(n)) information helps choose a set of final candidates and direct measurement helps final choice

Algorithm Growth Rates

- An algorithm's time requirements can be measured as a function of the problem size
  - Number of nodes in a linked list
  - Size of an array
  - Number of items in a stack
  - Number of disks in the Towers of Hanoi problem

Algorithm Growth Rates – 2



- Algorithm A requires time proportional to n^2
- Algorithm B requires time proportional to n

Algorithm Growth Rates – 3

- An algorithm's growth rate enables comparison of one algorithm with another
- Example
  - if, algorithm A requires time proportional to n^2, and algorithm B requires time proportional to n
  - algorithm B is faster than algorithm A
  - n^2 and n are growth-rate functions
  - Algorithm A is O(n^2) - order n^2
  - Algorithm B is O(n) - order n
- Growth-rate function f(n)
  - mathematical function used to specify an algorithm's order in terms of the size of the problem

Order-of-Magnitude Analysis and Big O Notation

(a)

	n					
Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Figure 9-3a A comparison of growth-rate functions: (a) in tabular form

Order-of-Magnitude Analysis and Big O Notation

(b)

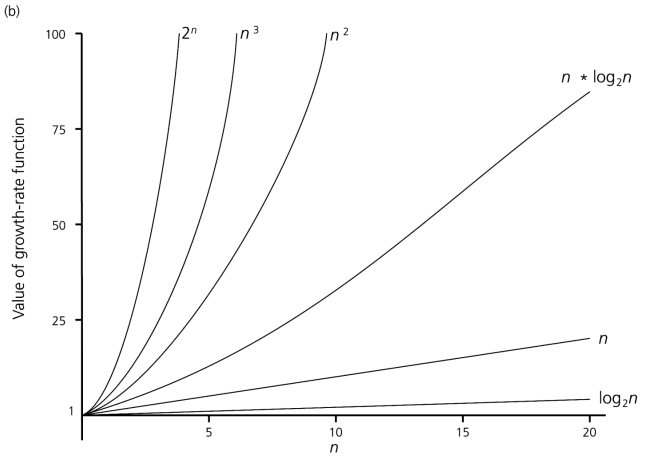


Figure 9-3b A comparison of growth-rate functions: (b) in graphical form

Order-of-Magnitude Analysis and Big O Notation

- Order of growth of some common functions
  - $O(C) < O(\log(n)) < O(n) < O(n * \log(n)) < O(n^2) < O(n^3) < O(2^n) < O(3^n) < O(n!) < O(n^n)$
- Properties of growth-rate functions
  - $O(n^3 + 3n)$ is $O(n^3)$: ignore low-order terms
  - $O(5\ f(n)) = O(f(n))$: ignore multiplicative constant in the high-order term
  - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

Keeping Your Perspective

- Only significant differences in efficiency are interesting
- Frequency of operations
  - when choosing an ADT's implementation, consider how frequently particular ADT operations occur in a given application
  - however, some seldom-used but critical operations must be efficient

Keeping Your Perspective

- If the problem size is always small, you can probably ignore an algorithm's efficiency
  - order-of-magnitude analysis focuses on large problems
- Weigh the trade-offs between an algorithm's time requirements and its memory requirements
- Compare algorithms for both style and efficiency

Sequential Search

- Sequential search
  - look at each item in the data collection in turn
  - stop when the desired item is found, or the end of the data is reached

```
int search(const int a[ ], int number_used, int target) {
      int index = 0; bool found = false;
      while ((!found) && (index < number_used)) {
            if (target == a[index])
                  found = true;
            else
                  Index++;
       }
      if (found)   return index;
      else    return  -1;
}
```

Efficiency of Sequential Search

- Worst case: O(n)
  - key value not present, we search the entire list to prove failure
- Average case: O(n)
  - all positions for the key being equally likely
- Best case: O(1)
  - key value happens to be first

The Efficiency of Searching Algorithms

- Binary search of a sorted array
  - Strategy
    - Repeatedly divide the array in half
    - Determine which half could contain the item, and discard the other half
  - Efficiency
    - Worst case: $O(\log_2 n)$
    - For large arrays, the binary search has an enormous advantage over a sequential search
      - At most 20 comparisons to search an array of one million items

Sorting Algorithms and Their Efficiency

- Sorting
  - A process that organizes a collection of data into either ascending or descending order
  - The sort key is the data item that we consider when sorting a data collection
- Sorting algorithm types
  - comparison based
    - bubble sort, insertion sort, quick sort, etc.
  - address calculation
    - radix sort

Sorting Algorithms and Their Efficiency

- Categories of sorting algorithms
  - An internal sort
    - Requires that the collection of data fit entirely in the computer's main memory
  - 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

Selection Sort

- Strategy
  - Place the largest (or smallest) item in its correct place
  - Place the next largest (or next smallest) item in its correct place, and so on
- Algorithm

```
for index=0 to size-2 {
    select min/max element from among A[index], …, A[size-1];
    swap(A[index], min);
}
```

- Analysis
  - worst case: $O(n^2)$, average case: $O(n^2)$
  - does not depend on the initial arrangement of the data

Selection Sort

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**

Bubble Sort

- Strategy
  - compare adjacent elements and exchange them if they are out of order
    - moves the largest (or smallest) elements to the end of the array
  - repeat this process
    - eventually sorts the array into ascending (or descending) order
- Analysis: worst case: $O(n^2)$, best case: $O(n)$

Bubble Sort – algorithm

```
for i = 1 to size - 1   do
    for index = 1 to size - i do
        if A[index] < A[index-1]
            swap(A[index], A[index-1]);
    endfor;
endfor;
```

Bubble Sort

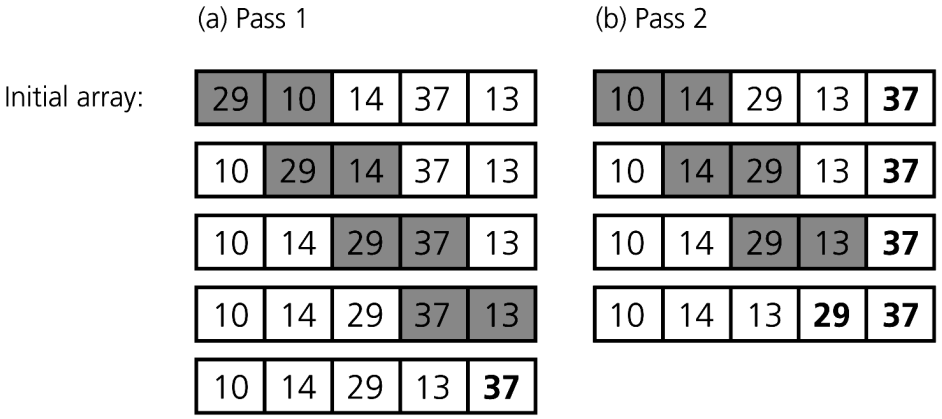


Figure 9-5

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

Insertion Sort

- Strategy
  - Partition array in two regions: sorted and unsorted
    - initially, entire array is in unsorted region
    - take each item from the unsorted region and insert it into its correct position in the sorted region
    - each *pass* shrinks unsorted region by 1 and grows sorted region by 1
- Analysis
  - Worst case: $O(n^2)$
    - Appropriate for small arrays due to its simplicity
    - Prohibitively inefficient for large arrays

Insertion Sort

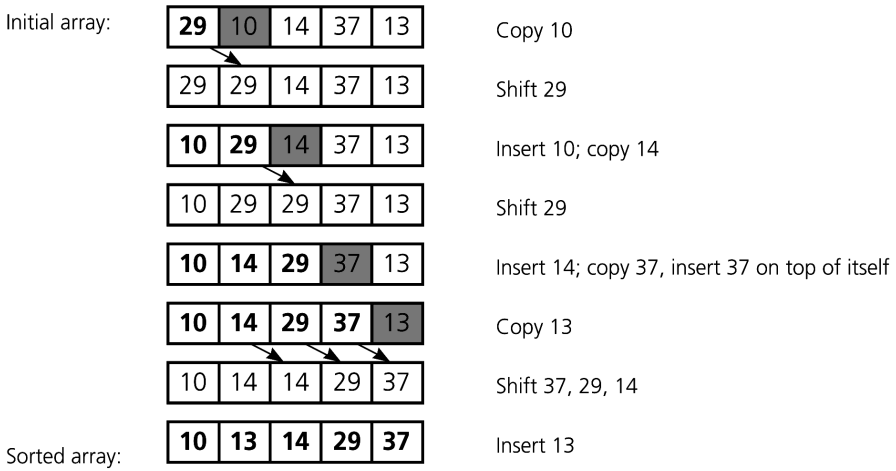


Figure 9-7 An insertion sort of an array of five integers.

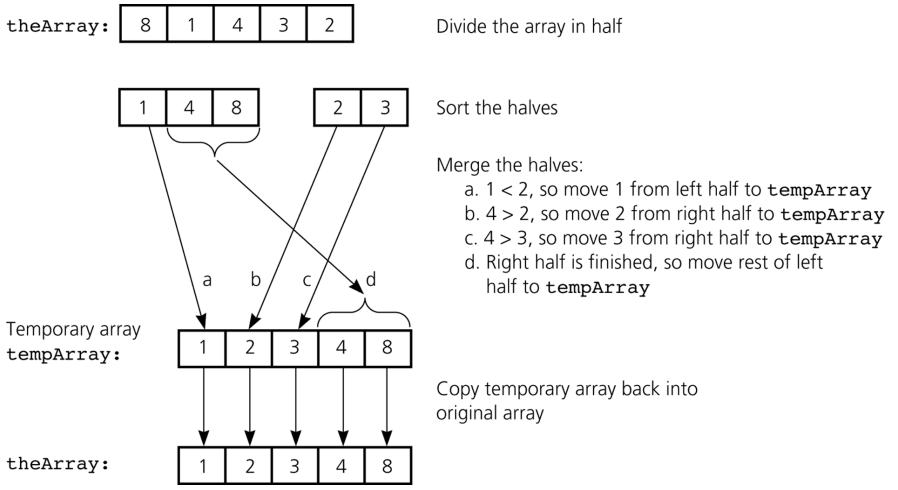
Mergesort

- A recursive sorting algorithm
- Performance is independent 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
  - divide-and-conquer approach

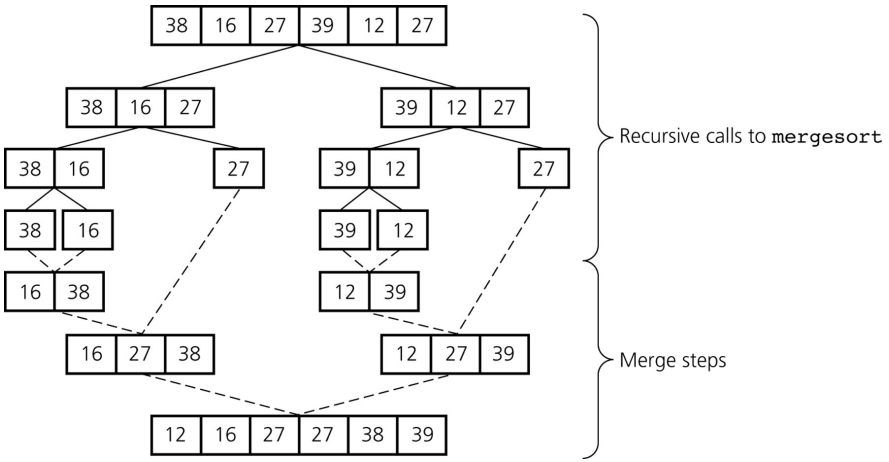
Mergesort – Algorithm

```
mergeSort(A,first,last) {
        if (first < last) {
                mid = (first + last)/2;
                mergeSort(A, first, mid);
                mergeSort(A, mid+1, last);
                merge(A, first, mid, last)
        }
}
```


Mergesort



Mergesort



Mergesort – Properties

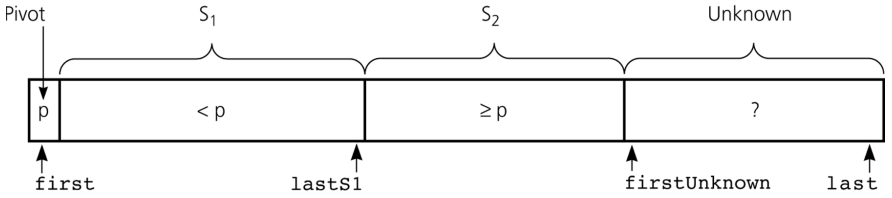
- Needs a temporary array into which to copy elements during merging
  - doubles space requirement
- Mergesort is *stable*
  - items with equal key values appear in the same order in the output array as in the input
- Advantage
  - mergesort is an extremely fast algorithm
- Analysis: worst / average case: O(n * log2n)

Quicksort

- A recursive divide-and-conquer algorithm
  - given a linear data structure A with n records
  - divide A into sub-structures S_1 and S_2
  - sort S_1 and S_2 recursively
- Algorithm
  - Base case: if |S|==1, S is already sorted
  - Recursive case:
    - divide A around a pivot value P into S_1 and S_2 , such that all elements of S_1<=P and all elements of S_2>=P
    - recursively sort S1 and S2 in place

Quicksort

- Partition()
  - (a) scans array, (b) chooses a pivot, (c) divides A around pivot, (d) returns pivot index
  - Invariant: items in S_1 are all less than pivot, and items in S_2 are all greater than or equal to pivot
- Quicksort()
  - partitions A, sorts S_1 and S_2 recursively



Quicksort – Pivot Partitioning

- Pivot selection and array partition are fundamental work of algorithm
- Pivot selection
  - perfect value: median of A[ ]
    - sort required to determine median (oops!)
    - approximation: If |A| > N, N==3 or N==5, use median of N
  - Heuristic approaches used instead
    - Choose A[first] OR A[last] OR A[mid] (mid = (first+last)/2) OR Random element
    - heuristics equivalent if contents of A[ ] randomly arranged

Quicksort – Pivot Partitioning Example

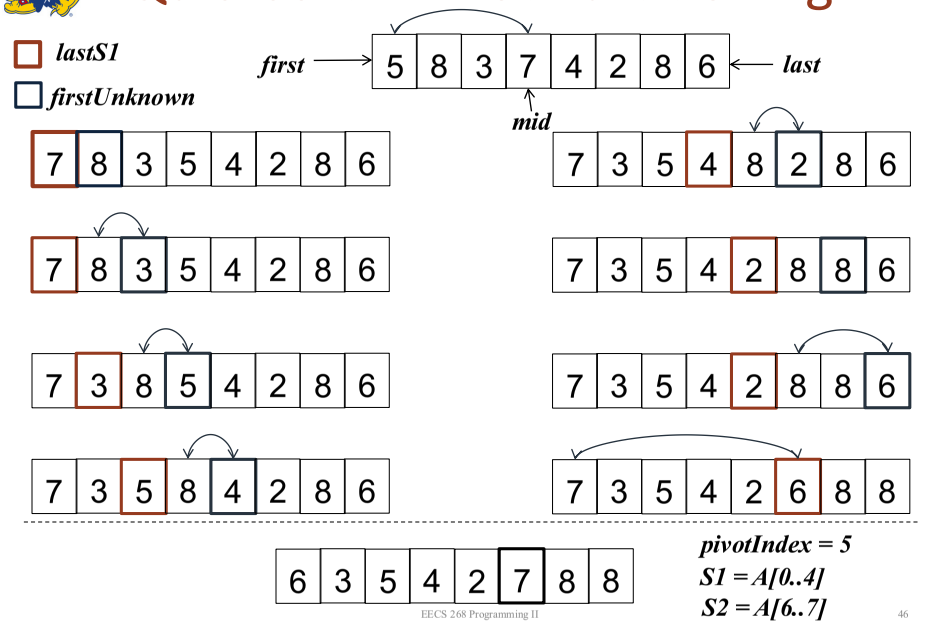
- A= [5,8,3,7,4,2,1,6], first =0, last =7
- A[first]: pivot = 5, A[last]: pivot = 6,
- A[mid]: mid =(0+7)/2=3, pivot = 7
- A[random()]: any key might be chosen
- A[medianof3]: median(A[first], A[mid], A[last]) is median(5,7,6) = 6
  - a sort of a fixed number of items is only O(1)
- Good pivot selection
  - computed in O(1) time and partitions A into roughly equal parts S1 and S2

Quicksort – Pivot Partitioning

- Middle element is pivot
- lastS1: index of last element of S1 partition
- firstUnknown: first element needing classification
  - if <p, then add to first partition by incrementing last S1 and swapping
  - incrementing firstUnknown expands partitioned sets either way
- Partitioning is an O(n) operation over A[ ]

```
int partition(A,first,last) {
    middle = (first+last)/2;
    pivot = A[middle];
    swap(A[middle],A[first]);
    lastS1 = first;
    firstUnknown = first+1;
    while( firstUnknown <= last ) {
        if (A[firstUnknown] < pivot) {
            lastS1++;
            swap(A[firstUnknown],A[lastS1]);
        }
        firstUnknown++;
    }
    swap(A[first],A[lastS1]);
    pivotIndex = lastS1;
    return(pivotIndex);
}
```


Quicksort – Pivot Partitioning



Quicksort – Analysis

- Best case
  - perfect partition at each level, $\log_2 n$ levels
  - O(n log n) total
- Average case
  - roughly equal partition
  - O(n log n)
- Worst case
  - S1 or S2 always empty
  - When the array is already sorted and the smallest item is chosen as the pivot
  - $O(n^2)$, n levels, rare as long is input is in random order

Quicksort – Analysis

- Partitioning and recursive call overhead is such that for |A| < 10 or so it is faster to simply use insertion sort
  - precise tipping point will vary with architecture
  - but, Quicksort is usually extremely fast in practice
- Not stable like Mergesort, but sorts in place
- Even if the worst case occurs, quicksort's performance is acceptable for moderately large arrays

Radix Sort

- Radix sort is a special kind of distribution sort that can efficiently sort data items using integer or other t element keys $(a_t a_{t-1} \ldots a_0)_m$ in a given radix (base) m
  - character string keys work as well; total order of all characters required
- Strategy
  - Treats each data element as a character string
  - Repeatedly organizes the data into groups according to the i^{th} character in each element

Radix Sort

- Basic idea
  - each key consists of t places, each holding one of m possible values
  - use m buckets and iterate the basic algorithm t times, each time using a different element of the key for sorting
  - iterate from least significant to most significant key position
  - 12345 – Five digit key, iterated over 10^0, 10^1, 10^2, 10^3, 10^4 using buckets 0-9 each time
  - FRED – Four character keys using capitol letters, iterated from right to left using 26 buckets A-Z each time
- 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)

Figure 9-21 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	$n * \log n$	$n * \log n$

Figure 9-22 Approximate growth rates of time required for eight sorting algorithms

Summary

- Order-of-magnitude analysis and Big O notation measure an algorithm's time requirement as a function of the problem size by using a growth-rate function
- To compare the efficiency of algorithms
  - examine growth-rate functions when problems are large
  - consider only significant differences in growth-rate functions

Summary

- Worst-case and average-case analyses
  - worst-case analysis considers the maximum amount of work an algorithm will require on a problem of a given size
  - average-case analysis considers the expected amount of work that an algorithm will require on a problem of a given size

Summary

- Worst case complexity of sorting algorithms

	Imput in Sorted Order	Input in Reverse Sorted Order
Bubble Sort	O(n)	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)
Quick Sort	$O(n^2)$	$O(n^2)$
Radix Sort	O(n)	O(n)

Summary

- Complexity of sorting algorithms for random data, most common case

	TB(n)	TW(n)	TA(n)
Bubble	O(n)	$O(n^2)$	$O(n^2)$
Insertion	O(n)	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge	O(n log n)	O(n log n)	O(n log n)
Quiksort	O(n log n)	$O(n^2)$	O(n log n)
Radix	O(n)	O(n)	O(n)

Summary

- Stability of sorting algorithms
  - stable sort preserves the input order of data items with identical keys
  - Thus, if input items x and y have identical keys, and x precedes y in the input data set, x will precede y in the output sorted data set
  - bubble, insertion, selection, merge, and radix are stable sorting algorithms