Comparison of Algorithms

- How do we compare the efficiency of different algorithms?
- Comparing execution time: Too many assumptions, varies greatly between different computers
- Compare number of instructions: Varies greatly due to different languages, compilers, programming styles...

Big-O Notation

- The best way is to compare algorithms by the amount of work done in a critical loop, as a function of the number of input elements (N)
- **Big-O:** A notation expressing execution time (complexity) as the term in a function that increases most rapidly relative to N
- Consider the *order of magnitude* of the algorithm

Common Orders of Magnitude

- $O(1)$: Constant or *bounded* time; not affected by N at all
- $O(\log_2 N)$: Logarithmic time; each step of the algorithm cuts the amount of work left in half
- $O(N)$: Linear time; each element of the input is processed
- $O(N \log_2 N)$: $N \log_2 N$ time; apply a logarithmic algorithm N times or vice versa

Common Orders of Magnitude (cont.)

- $O(N^2)$: Quadratic time; typically apply a linear algorithm N times, or process every element with every other element
- $O(N^3)$: Cubic time; naive multiplication of two NxN matrices, or process every element in a three-dimensional matrix
- $O(2^N)$: Exponential time; computation increases dramatically with input size

What About Other Factors?

- Consider $f(N) = 2N^4 + 100N^2 + 10N + 50$
- We can ignore $100N^2 + 10N + 50$ because $2N^4$ grows so quickly
- Similarly, the 2 in $2N^4$ does not greatly influence the growth
- The final order of magnitude is $O(N^4)$
- The other factors may be useful when comparing two very similar algorithms

Elephants and Goldfish

- Think about buying elephants and goldfish and comparing different pet suppliers
- The price of the goldfish is trivial compared to the cost of the elephants
- Similarly, the growth from $100N^2 + 10N + 50$ is trivial compared to $2N^4$
- The smaller factors are essentially noise

Example: Phone Book Search

- Goal: Given a name, find the matching phone number in the phone book
- Algorithm 1: Linear search through the phone book until the name is found
- Best case: O(1) (it's the first name in the book)
- Worst case: $O(N)$ (it's the final name)
- Average case: The name is near the middle, requiring $N/2$ steps, which is $O(N)$

Example: Phone Book Search (cont.)

Algorithm 2: Since the phone book is sorted, we can use a more efficient search

1) Check the name in the middle of the book
2) If the target name is less than the middle name, search the first half of the book
3) If the target name is greater, search the last half
4) Continue until the name is found

Example: Phone Book Search (cont.)

Algorithm 2 Characteristics:

- Each step reduces the search space by half
- Best case: O(1) (we find the name immediately)
- Worst case: $O(\log_2 N)$ (we find the name after cutting the space in half several times)
- Average case: $O(\log_2 N)$ (it takes a few steps to find the name)

Example: Phone Book Search (cont.)

Which algorithm is better?

- For very small *N*, algorithm may be faster
- For target names in the very beginning of the phone book, algorithm 1 can be faster
- Algorithm 2 will be faster in every other case
- Success of algorithm 2 relies the fact that the phone book is sorted
  - Data structures matter!

Sorting Revisited

- Sorting is a very common and useful operation
- Efficient sorting algorithms can have large savings for many applications
- The algorithms are evaluated on:
  - The number of comparisons made
  - The number of times data is moved
  - The amount of additional memory used

Sorting Efficiency

- Worst Case: The data is in reverse order
- Average Case: Random data, may be somewhat sorted already
- Best Case: The array is already sorted
- Typically, average and worst case performance are similar, if not identical
- For many algorithms, the best case is also the same as the other cases

Straight Selection Sort

1) Set “current” to the first index of the array
2) Find the smallest value in the array
3) Swap the smallest value with the value in current
4) Increment current and repeat steps 2–4 until the end of the array is reached

	(a) values	(b) values	(c) values	(d) values	(e) values
[0]	126	1	1	1	1
[1]	43	43	26	26	26
[2]	26	26	43	43	43
[3]	1	126	126	126	113
[4]	113	113	113	113	126

Figure 12.1 Example of straight selection sort (sorted elements are shaded)

Analyzing Selection Sort

- A very simple, easy-to-understand algorithm
- *N* iterations are performed
- Iteration *I* checks *N – I* items to find the next smallest value
- There are $N * (N - 1)/2$ comparisons total
- Therefore, selection sort is O(N^2)
- Even in the best case, it’s still O(N^2)

Bubble Sort

1) Set “current” to the first index of the array
2) For every index from the end of the list to 1, swap adjacent pairs of elements that are out of order
3) Increment current and repeat steps 2–3
4) Stop when current is at the end of the array

	(a) values	(b) values	(c) values	(d) values	(e) values
[0]	36	6	6	6	6
[1]	24	36	10	10	10
[2]	10	24	36	12	12
[3]	6	10	24	36	24
[4]	12	12	12	24	36

Figure 12.3 Example of bubble sort (sorted elements are shaded)

Bubble Sort

- The name comes from how smaller elements “bubble up” to the top of the array
- The inner loop compares values [index] < values [index-1], and swaps the two values if it evaluates to true
- The smallest value is brought to the front of the unsorted portion of the array during iteration

Insertion Sort

- Acts like inserting elements into a sorted array, including moving elements down if necessary
- Uses swapping (like Bubble Sort) to find the correct position of the next item

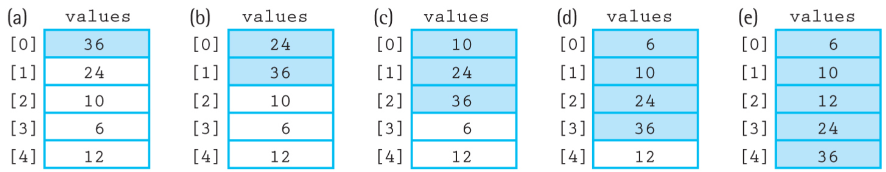

Figure 12.5 Example of the insertion sort algorithm

Analyzing Bubble Sort

- Takes *N-1* iterations, because the last iteration puts two values in order
- Each iteration *I* performs *N-I* comparisons
- Bubble sort is therefore $O(N^2)$
- It may perform several swaps per iteration
- Is the best case better? An already-sorted array needs only 1 iteration, so the base case is $O(N)$

Analyzing Insertion Sort

- $O(N^2)$, like the previous sorts
- Best Case: $O(N)$, since only one comparison is needed and no data is moved
- $O(N^2)$ is not good enough when sorting large sets of data!

$O(N \log_2 N)$ Sorts

- Sorting a whole array is $O(N^2)$ with those sorts
- Splitting the array in half, sorting it, and then merging the two arrays is $(N/2)^2 + (N/2)^2$
- This “divide-and-conquer” approach can then be applied to each half, giving $O(N \log_2 N)$ sort