Sorting

After studying this chapter, you should be able to

- Design and implement the following sorting algorithms:
  - Straight selection sort
  - Bubble sort (two versions)
  - Insertion sort
  - Merge sort
  - Quick sort
  - Heap sort
  - Radix sort
  - Parallel merge sort
- Compare the efficiency of the sorting algorithms, in terms of Big-O and space requirements

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

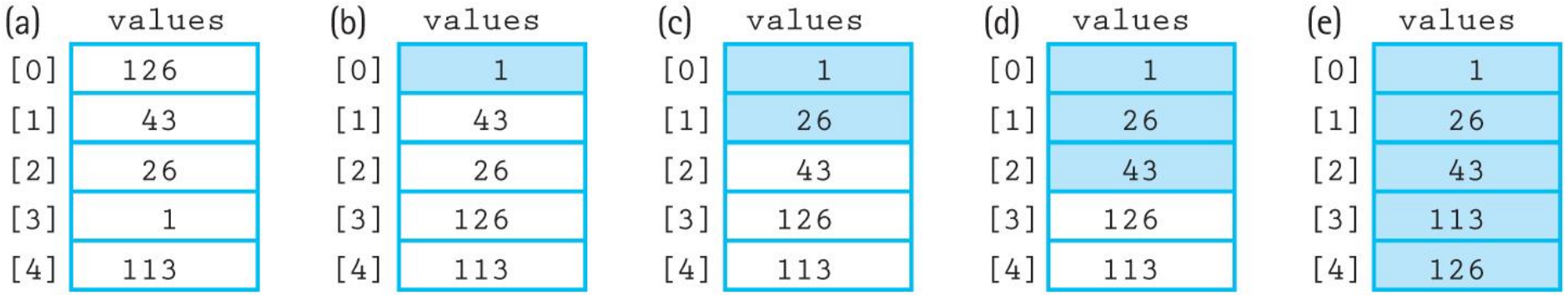

Figure 12.1 Example of straight selection sort (sorted elements are shaded)

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

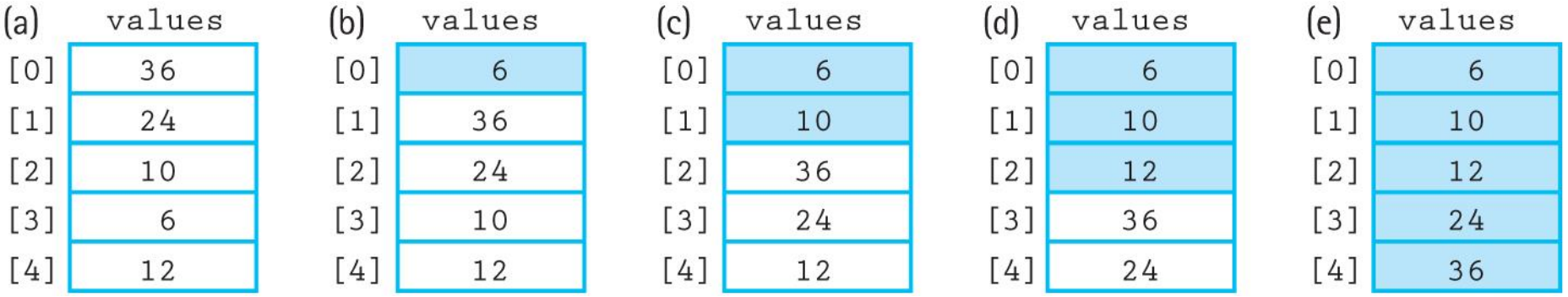

Figure 12.3 Example of bubble sort (sorted elements are shaded)

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

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 $O(N)$

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

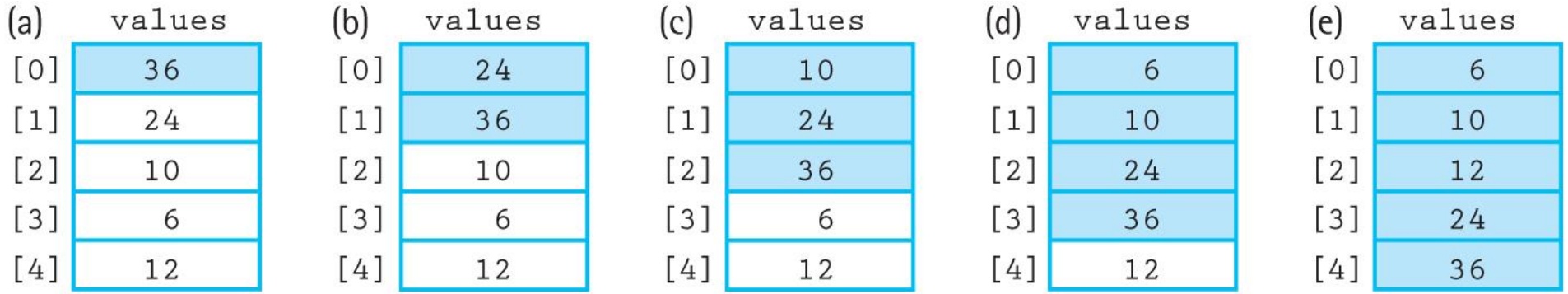

Figure 12.5 Example of the insertion sort algorithm

Analyzing 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

Merge Sort

1) Split the array in half
2) Recursively MergeSort the two halves
3) Merge the two halves
4) Base Case: Arrays of <2 elements are already sorted

Merging

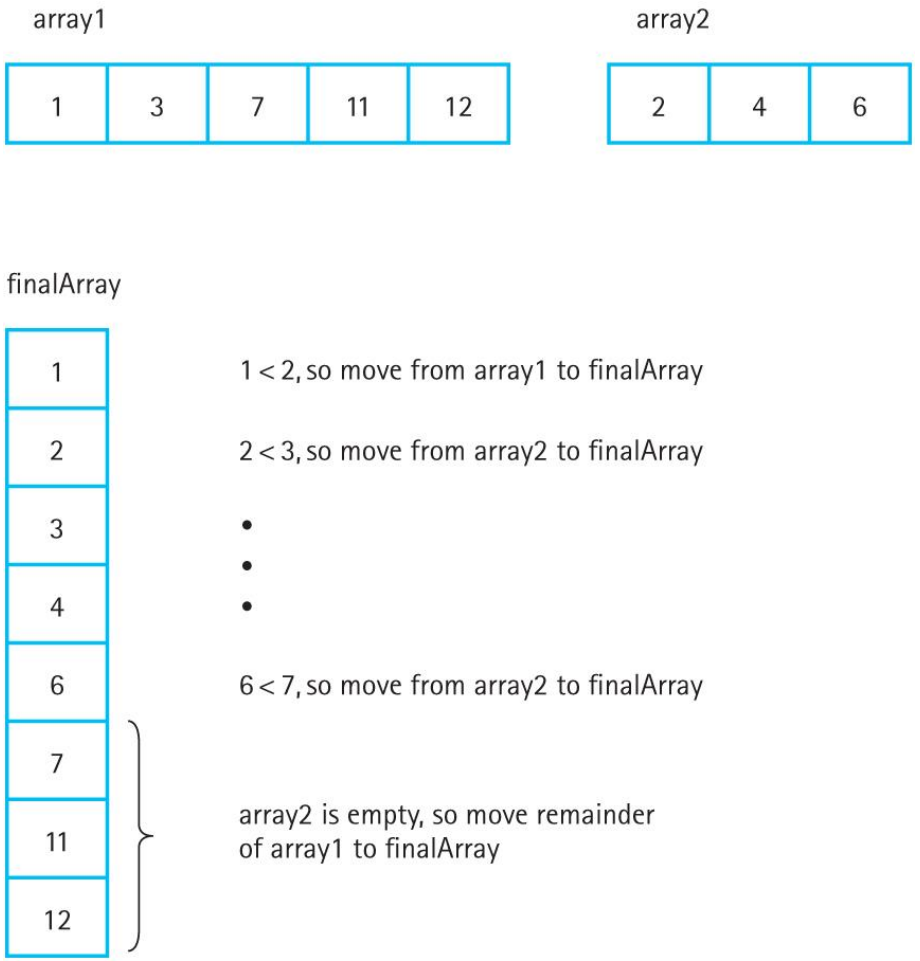

Figure 12.8 Strategy for merging two sorted arrays

Merging (cont.)

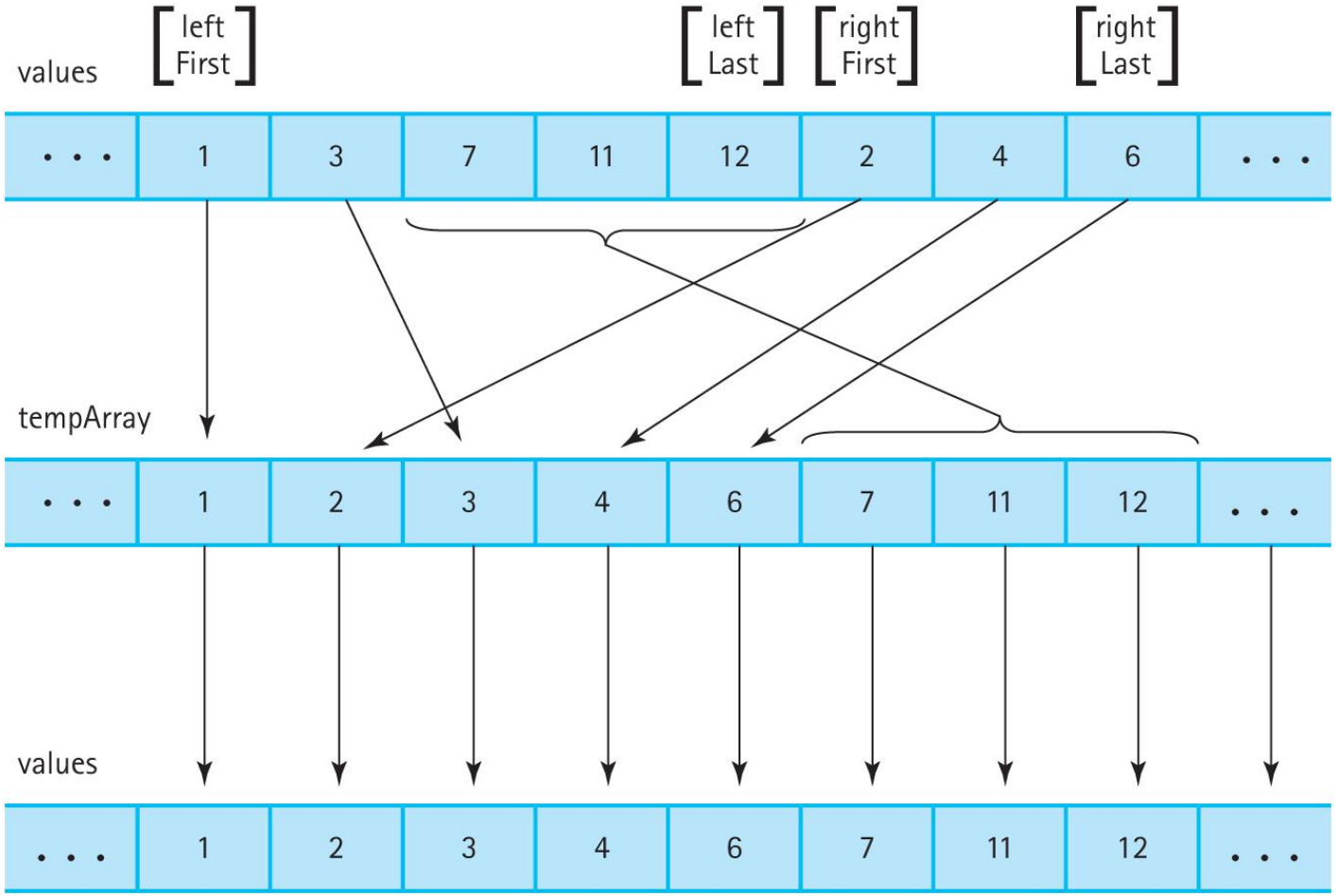

Figure 12.10 Merging sorted halves

Merging Two Arrays

- Simultaneously walk through both arrays
- If itemA < itemB, copy itemA into the temp array and get the next item from A
- If itemB < itemA, copy itemB into the temp array and get the next item from B
- Stop when one of the arrays is empty; copy any remaining elements from the other array
- Copy the temp array into the original array

Analyzing Merge Sort

- The array is split into smaller and smaller arrays until it is finally split into 1- or 0-element arrays
- Merging all of these 1-element arrays into 2-element arrays is $O(N)$
- This $O(N)$ merge repeats for each split, which happens $\log_2 N$ times for $O(N \log_2 N)$ total
- However, merge sort requires $O(N)$ additional memory to perform the merge

Merging Levels

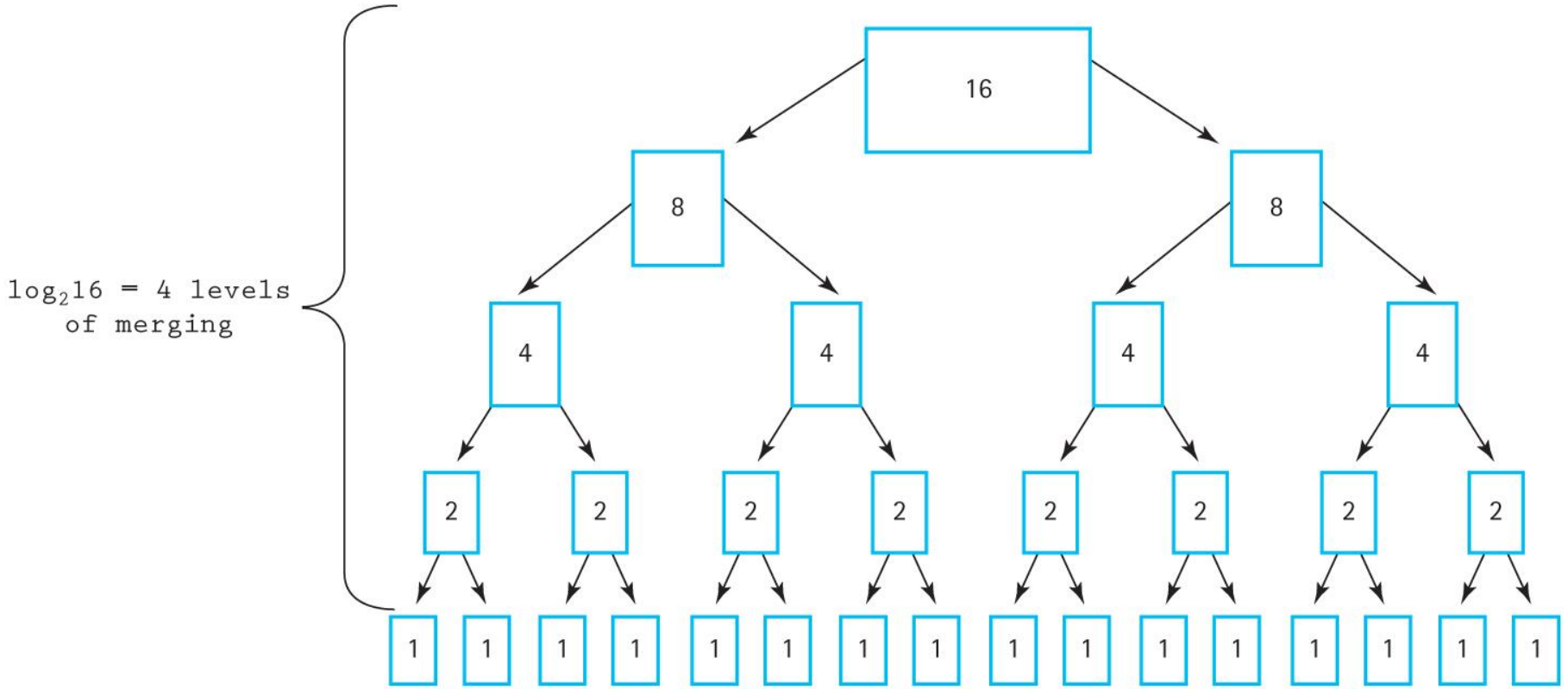

Figure 12.11 Analysis of the function MergeSort with N = 16

Quick Sort

- Choose a value from the array
- Use this value to split the array into:
  - All the values that are less than or equal to the split
  - All the values that are greater than the split
- Recursively quick sort the two subarrays

Analyzing Quick Sort

- If the arrays are evenly split, $\log_2 N$ splits are used and the algorithm is $O(N \log_2 N)$
- If the array is not split evenly, the algorithm approaches $O(N^2)$
- Quick Sort may be inefficient if the data is already mostly sorted
- $O(\log_2 N)$ to $O(N)$ additional memory needed, depending on the splits

Heap Sort

- Turns the array into a heap, then places the root at the last unsorted slot of the array until the heap is depleted
- Uses heap operations (ReheapDown) to find the next largest element in the heap
- Originally seen in Chapter 9

Analyzing Heap Sort

- The overhead for building the heap is significant, $O(N)$, and thus heap sort should be avoided for small arrays
- ReheapDown is $O(\log_2 N)$ and is called N times
- The overall performance is $O(N \log_2 N)$
- The initial order of the elements doesn't matter
- No additional space is needed

Other Considerations

- Efficiency is not the only thing to consider when choosing which sorting algorithm to use
- Big-O ignores constants and other factors, and in some situations $O(N^2)$ may grow slower than $O(N \log_2 N)$
- Note: $O(N \log_2 N)$ is the theoretical limit for sorting; it can't get better than that!
  - But maybe we can improve the coefficients

When N Is Small

- $O(N^2)$ sorts often have low overhead and few actions besides comparison
- $O(N \log_2 N)$ sorts usually have more overhead and more actions, such as merging
- When N is small enough, an $O(N^2)$ sort might be faster than an $O(N \log_2 N)$ sort

Eliminating Calls to Functions

- Every function call adds some overhead
- But replacing each function call with the body of the function makes code less clear
- The **`inline`** keyword suggests to the compiler to replace a function call with the function body, giving the best of both worlds
- This is useful for functions like Swap that are short and used frequently

Programmer Time

- Someone has to take the time to design, implement, and test the sorting algorithms
- More complex algorithms are more likely to have errors
- Fortunately, we can often rely on standard libraries and packages that have highly optimized and well-debugged implementations

Space Considerations

- MergeSort requires an additional *N* memory
- Depending on the implementation and the data, QuickSort may use O(*N*) memory
- HeapSort is more algorithmically complex but uses no additional memory
- Consider the memory constraints of the system and the size of the input when choosing

Keys and Stability

- A record may contain multiple keys, such as a student's name, ID number, and major
- Some of these keys may not be unique
- What if we want a list sorted by major, but have alphabetical order of names within each major?
- Sort twice, first by name and then by major
- This only works if the sort is *stable*

Stable Sorting Algorithms

- A **stable sort** is a sorting algorithm that preserves the order of duplicate keys
- Heap sort is the only algorithm so far that is inherently unstable
- Some algorithms may be unstable depending on their implementation

Stable Sort Example

- Start with a list of students:
  - **(ID, Major, Last name)**
  - (01, CHM, Davis)
  - (02, CSC, Fernandez)
  - (03, CHM, Patil)
  - (04, CSC, Albaf)
  - (05, BIO, Hart)
- Goal: sorted alphabetically within majors

Stable Sort Example (cont.)

- Sort by Last Name:
  - (04, CSC, Albaf)
  - (01, CHM, Davis)
  - (02, CSC, Fernandez)
  - (05, BIO, Hart)
  - (03, CHM, Patil)

Stable Sort Example (cont.)

- Sort by Major:
  - (05, BIO, Hart)
  - (01, CHM, Davis)
  - (03, CHM, Patil)
  - (04, CSC, Albaf)
  - (02, CSC, Fernandez)

Stable Sort Example (cont.)

- An unstable sort would have come up with a different result, such as:
  - (05, BIO, Hart)
  - (03, CHM, Patil)
  - (01, CHM, Davis)
  - (04, CSC, Albaf)
  - (02, CSC, Fernandez)
- The majors are correct, but last names aren’t

Sorting with Pointers

- Moving around large objects is inefficient
- Instead, have an array of pointers to those large objects, and have the sort move the pointers
- The objects don't move but can still be accessed in sorted order

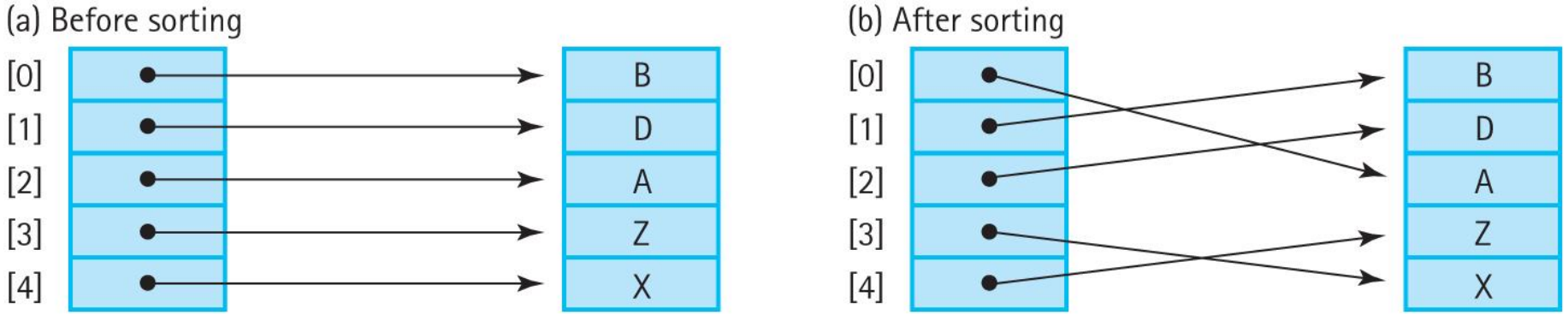

Figure 12.12 Sorting arrays with pointers (a) Before sorting (b) After sorting

Caching

- Not all computer memory is equal, even though we like to pretend it is
- It takes as much time to access a memory cell as for the computer to execute 100 instructions
- The processor uses a *cache*, a small store of very fast memory, to access a small amount of memory quickly

Caching (cont.)

- The cache can store a very limited amount of data and code
- For example, it may store only a few adjacent list elements, and accessing other list elements requires fetching those elements from memory
- An algorithm that makes good use of the cache will be faster than other algorithms that don’t

Sorting Algorithms and Caching

- Bubble Sort has predictable memory access patterns, and only compares adjacent elements, which complements the cache's behavior
- Selection Sort must search the whole list for the next smallest element, which will not work well with the cache
- An array of pointers to objects scattered throughout memory defeats the cache, since there's no way to predict the next access

Radix Sort

- Radix sort is not a comparison sort; it does not sort by comparing items against each other
- The sort considers each position in the key and groups keys by their *radix*
- **Radix:** The number of possibilities for each position; the digits in a number system
  - The lowercase letters have a radix of 26
  - The base-10 digits have a radix of 10

Radix Sort Algorithm

- Consider sorting three-digit positive integers
- We have 10 queues, one for each digit (0–9)
- All the items with a 0 in the ones position are inserted into queue[0], etc.
- Once all numbers have been sorted into the queues, collect the values from the queues
- Repeat these steps with the tens and hundreds positions

Radix Sort Example

(a) Queues after 1st pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
800	761	762		124		976			999
100	001	432							
		402							

(b) Queues after 2nd pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
800		124	432			761	976		999
100						762			
001									
402									

(c) Queues after 3rd pass

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
001	100			402			761	800	976
	124			432			762		999

Figure 12.14 Queues after each pass (a) Queues after 1st pass (b) Queues after 2nd pass (c) Queues after 3rd pass

Radix Sort Examples

Original array	*Array after 1st pass*	*Array after 2nd pass*	*Array after 3rd pass*
762	800	800	001
124	100	100	100
432	761	001	124
761	001	402	402
800	762	124	432
402	432	432	761
976	402	761	762
100	124	762	800
001	976	976	976
999	999	999	999

Figure 12.13 Array after each pass

Radix Sort’s ItemType

- Radix Sort must be able to extract a number for each position in the key
- This varies between types and is handled by the client code by changing ItemType
- Radix Sort needs to know the length of the keys and the radix

Analyzing Radix Sort

- Each item (*N*) is processed once for each position in the key (*P*)
- Radix sort is therefore O(*N***P*), but since *P* is likely small relative to *N*, it's simply O(*N*)
- Extra memory is needed for the queues
- No extra memory is needed if linked lists and linked queues are used; values can be moved from list to queue by changing the pointers

Parallel Merge Sort

- Sometimes $O(N \log_2 N)$ just isn't enough
- **Parallel processing** performs multiple operations simultaneously, unlike the usual **serial** (or sequential) **processing**
- Merge sort divides the work into independent parts, which makes it an ideal candidate for splitting among several *threads*

A Thread for Every Merge

- For the first attempt at parallelizing Merge Sort, create a new thread for every recursive call
- Each parent thread waits for the two child threads it spawns to finish, then merges the now-sorted arrays
- This approach causes incredible slowdowns and will crash the program on large inputs
- Each thread has a lot of overhead

Chunks

- At a certain array size, threads cost more than they could possible save
- When the array is larger than a particular size, spawn new threads; otherwise use regular sequential merge sort
- The **chunk size** is how large arrays must be in order to create more threads
- This is passed as a parameter to Merge Sort

Different Chunk Sizes

Table 12.3 Performance of Parallel Merge Sort

Chunk Size	Number of Threads	Time (Seconds)
20,000,000	1	8.2
10,000,000	2	4.2
5,000,000	4	2.4
2,500,000	8	1.5
1,250,000	16	1.5
625,000	32	1.5
312,500	64	1.5
100,000	200	1.5
25,000	800	1.6

Table 12.3 Performance of Parallel Merge Sort

Note that the results are machine dependent. This is a quad-core machine, which can handle at most eight threads at once

Parallel Gains

- From the table, spawning eight threads greatly increased performance
- But shouldn’t the execution time be 1/8th of the time for a single thread?
- No, because merging the arrays is still serial

Parallel Merging

- Could merging the arrays be made parallel?
- We could try splitting the arrays similar to how quick sort does, by finding a split point and having each thread merge all the values below or above the split value
- But, like quick sort, this could cause very uneven splits and be even more inefficient
- Parallel merging is very complex with little gain

Avoiding Parallel Access

- Merge Sort works well in parallel because each divided part is independent
- None of the threads access parts of the array that another thread is using
- This kind of interaction can lead to incorrect and unpredictable results

Is Parallelism Worth It?

- Parallel merge sort, in this example, was 5.5 times faster than sequential merge sort
- Larger data sets and faster, more parallel-capable machines will have even better gains
- But parallel computing has complex code and complex bugs
- Overall, parallelism is worth it, if it is implemented correctly

The End!