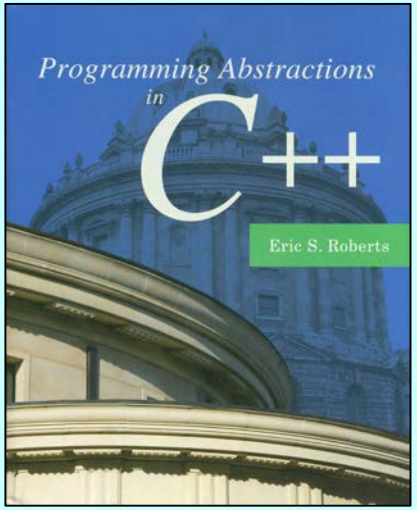
CHAPTER 10

Algorithmic Analysis

Without analysis, no synthesis.

—Friedrich Engels, *Herr Eugen Dühring's Revolution in Science*, 1878



Recursion to the Rescue

- As long as arrays are small, selection sort is a perfectly workable strategy. Even for 10,000 elements, the average running time of selection sort is just over a second.
- The quadratic behavior of selection sort, however, makes it less attractive for the very large arrays that one encounters in commercial applications. Assuming that the quadratic growth pattern continues beyond the timings reported in the table, sorting 100,000 values would require two minutes, and sorting 1,000,000 values would take more than three hours.
- The computational complexity of the selection sort algorithm, however, holds out some hope:
  - Sorting twice as many elements takes four times as long.
  - Sorting half as many elements takes only one fourth the time.
  - Is there any way to use sorting half an array as a subtask in a recursive solution to the sorting problem?

Merge sort

- Classic Divide and Conquer:
  - Split the data in half
  - Sort each half recursively
  - Merge the results together

The Merge Sort Idea

1. Divide the vector into two halves: **`v1`** and **`v2`**.
2. Sort each of **`v1`** and **`v2`** recursively.
3. Clear the original vector.
4 Merge elements into the original vector by choosing the smallest element from **`v1`** or **`v2`** on each cycle.

`vec`

236	259	361	367	503	659	809	838	946	987
0	1	2	3	4	5	6	7	8	9

`v1`

0	1	2	3	4

`v2`

0	1	2	3	4

The Merge Sort Implementation

```
/*
 * The merge sort algorithm consists of the following steps:
 *
 * 1. Divide the vector into two halves.
 * 2. Sort each of these smaller vectors recursively.
 * 3. Merge the two vectors back into the original one.
 */

void merge_sort(Vector<int> & vec) {
   int n = vec.size();
   if (n <= 1) return;
   Vector<int> v1;
   Vector<int> v2;
   for (int i = 0; i < n; i++) {
      if (i < n / 2) {
         v1.add(vec[i]);
      } else {
         v2.add(vec[i]);
      }
   }
   merge_sort(v1);
   merge_sort(v2);
   vec.clear();
   merge(vec, v1, v2);
}
```

The Merge Sort Implementation

```
/*
 * Function: merge
 * ---------------
 * This function merges two sorted vectors (v1 and v2) into the
 * vector vec, which should be empty before this operation.
 * Because the input vectors are sorted, the implementation can
 * always select the first unused element in one of the input
 * vectors to fill the next position.
 */

void merge(Vector<int> & vec, Vector<int> & v1, Vector<int> & v2) {
   int n1 = v1.size();
   int n2 = v2.size();
   int p1 = 0;
   int p2 = 0;
   while (p1 < n1 && p2 < n2) {
      if (v1[p1] < v2[p2]) {
         vec.add(v1[p1++]);
      } else {
         vec.add(v2[p2++]);
      }
   }
   while (p1 < n1) vec.add(v1[p1++]);
   while (p2 < n2) vec.add(v2[p2++]);
}
```

The Complexity of Merge Sort

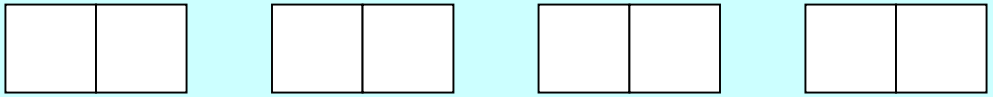
Sorting 8 items

requires

Two sorts of 4 items

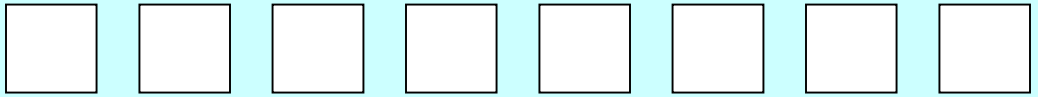
which requires

Four sorts of 2 items



which requires

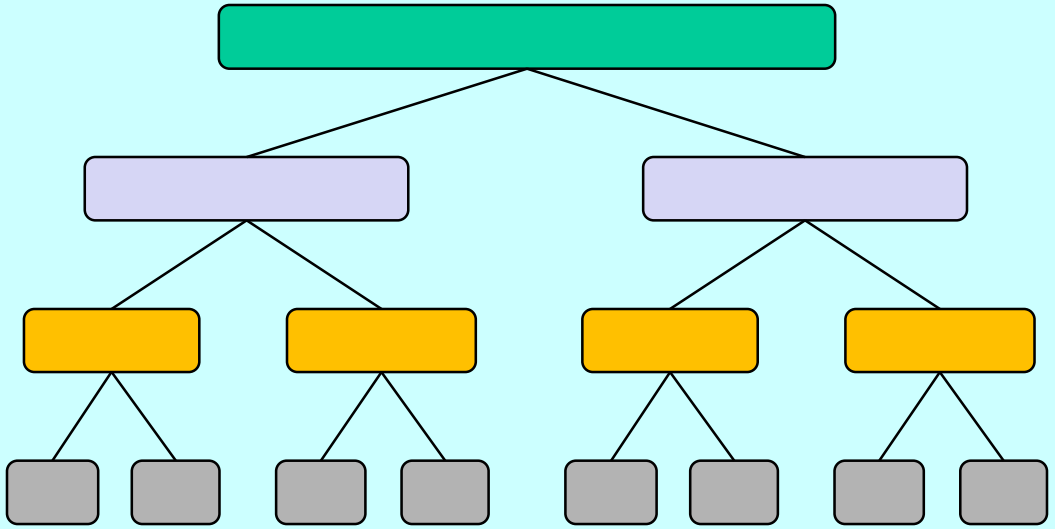
Eight sorts of 1 item



The work done at each level (*i.e.*, the sum of the work done by all the calls at that level) is proportional to the size of the vector. The running time is therefore proportional to *N* times the number of levels.

Analysis of Merge-Sort

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



- **The height h of the merge-sort tree is $O(\log n)$**
  - **at each recursive call we divide in half the sequence,**
- **The overall amount or work done at the nodes of depth i is $O(n)$**
  - **we partition and merge 2^i sequences of size $n/2^i$**
  - **we make 2^{i+1} recursive calls**
- **Thus, the total running time of merge-sort is $O(n \log n)$**

How Many Levels Are There?

- The number of levels in the merge sort decomposition is equal to the number of times you can divide the original vector in half until there is only one element remaining. In other words, what you need to find is the value of k that satisfies the following equation:

$$1 = N / \underbrace{2 / 2 / 2 / 2 \cdots / 2}_{k \text{ times}}$$

- You can simplify this formula using basic mathematics:

$$1 = N / 2^k$$
$$2^k = N$$
$$k = \log_2 N$$

- The complexity of merge sort is therefore $O(N \log N)$.

Comparing N^2 and $N \log N$

- The difference between $O(N^2)$ and $O(N \log N)$ is enormous for large values of N, as shown in this table:

N	N^2	$N \log_2 N$
10	100	33
100	10,000	664
1,000	1,000,000	9,966
10,000	100,000,000	132,877
100,000	10,000,000,000	1,660,964
1,000,000	1,000,000,000,000	19,931,569

- Based on these numbers, the theoretical advantage of using merge sort over selection sort on a vector of 1,000,000 values would be a factor of more than 50,000.

Standard Complexity Classes

- The complexity of a particular algorithm tends to fall into one of a small number of standard complexity classes:

constant	$O(1)$	Finding first element in a vector
logarithmic	$O(\log N)$	Binary search in a sorted vector
linear	$O(N)$	Summing a vector; linear search
$N \log N$	$O(N \log N)$	Merge sort
quadratic	$O(N^2)$	Selection sort
cubic	$O(N^3)$	Obvious algorithms for matrix multiplication
exponential	$O(2^N)$	Tower of Hanoi solution

- In general, theoretical computer scientists regard any problem whose complexity cannot be expressed as a polynomial as ***intractable***.

Graphs of the Complexity Classes