1: The role of algorithms
2: Getting started
3: Growth of functions
4: Recurrences
5: Probabilistic analyses
6: Heapsort
7: Quicksort
8: Sorting in linear time
9: Medians and other statistics
10: Elementary data structures
11: Hash tables
12: Binary search trees
13: Red-black trees
20: Fibonacci heaps
21: Data structures for disjoint sets
22: Elementary graph algos
23: Minimum spanning trees
24: Single-source shortest paths
25: All-pairs shortest paths
26: Maximum flow
30: Polynomials and the FFT
31: Number-theoretic algos
32: String matching
33: Computational geometry
34: NP-completeness
App. A: Summations
App. B: Sets, etc.
App. C: Counting and probability
An outline of the book
Definitions and terms:

- Average case analysis
- Worst case analysis
- Cost of an algorithm
- Heuristic
- Algorithm
- Procedure

We will also cover some material that more specifically covers parallel algorithms.

We will cover some, but not all, the appendix material.

We will cover some, but certainly not all, the chapters in this book.
<table>
<thead>
<tr>
<th>Duration</th>
<th>1 Year</th>
<th>2.921 × 10^12</th>
<th>1 Month</th>
<th>2.5 × 10^10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.1536 × 10^13</td>
<td>1 Week</td>
<td>8.64 × 10^7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3600</td>
<td>1 Day</td>
<td>86400</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 Hour</td>
<td>3600</td>
<td></td>
</tr>
</tbody>
</table>

Computer architecture matters

But of course not all ops are the same

1 GHz = 10^9 ops per second
By dividing by the boldface entry:

```
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>6</td>
<td>2.0</td>
<td>9</td>
<td>2.0</td>
<td>9</td>
<td>2.0</td>
<td>9</td>
<td>2.0</td>
</tr>
<tr>
<td>2.0</td>
<td>2</td>
<td>2.0</td>
<td>2</td>
<td>2.0</td>
<td>2</td>
<td>2.0</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>2.0</td>
<td>2</td>
<td>2.0</td>
<td>2</td>
<td>2.0</td>
<td>2</td>
<td>2.0</td>
<td>2</td>
<td>2.0</td>
</tr>
</tbody>
</table>
```

The second part of the table normalizes against an exponential algorithm, assuming a $O(n \log n)$ representation of $n$. The numbers in the table are normalized with the base in the first column, numbers (times $10^6$) represent the size of $n$ seconds in a 28-day month, so one month is $2.4192 \cdot 10^9$ ops. How big a problem is this? The following second, there are $28 \cdot 24 \cdot 3600 = 2.4192 \cdot 10^9$ ops.

Assume a computer does "ops" at 1 GHz, that is, at $O(n \log n)$ per second, there are $10^9$ seconds in a 28-day month, so one month is $2.4192 \cdot 10^9$ ops. How big a problem is this?
Definition 1.2. A Random Access Machine, or RAM, is a PRAM with a single processor.

Processors have access to the entire global memory. Processors may be idle in a given time step rather than execute the operation. But the control unit, Processors execute the same operation with each time step, under the control of the control unit. Processors execute the same operation with each time step, under the control of the control unit.

We assume that all processors execute the same operation with each time step, under the control of the control unit.

An interconnection between the processors and the global memory.

A global memory (assumed to be relatively large compared to local memory).

An unbounded number of processors each with its own (assumed relatively small) local memory.

A control unit.

Definition 1.1. A Parallel Random Access Machine, or PRAM, comprises

1.3 What are we measuring?
Algorithm analysis is therefore only a guide to what ought to be done in real life, not a prescription for
what must be done.

On the more realistic machines, optimization problem on the RANN or PRANN becomes a very complicated multivariable optimization problem, Second, you probably can’t do the analysis on the more complicated machine because the single-variable algorithm on this idealized machine then you probably can’t do the analysis on a more complicated machine. First, if you can’t analyze an algorithm on the last machine that actually behaved like this was the VAX. For the most part, however, the algorithm

Only one word is desired, etc.

Memory access is taken to be of a single data item at a time—no cache line fetches of doublewords when
etc. Memory is taken to be a single, large, flat, memory—no cache, no memory banks, no virtual memory,
etc. Memory is assumed to be executing a single instruction, start to finish, at a time—no pipelining, no overlapped execution,

Caveats: The definition of a PRANN (and of a RANN) ignores most of the lower-level computer architecture.
Implementation issues can matter and are usually not counted in the asymptotics.

Space can matter and is usually not counted in the asymptotics.

The constants matter as well as the asymptotics.

Worst case versus average case analysis.

operations, and that this will complicate the problem of analysis.

We admit that not all algorithms for solving the same problem necessarily will involve the same basic

memory accesses for databases?

Let’s count the right things (comparisons for sorting, floating point multiplications for scientific computation).

So in doing algorithm analysis:
Network sorts: Fast; but require hardware.

Merge sort: Good worst case; good average case; but takes extra space.

Heap sort: Good worst case; same average case; no additional memory; makes random memory references.

Quick sort: Bad worst case; good average case; no additional memory; uses mostly local references.

Consider sorting algorithms.

1.4 Measuring the right things is not always trivial.
Multiply by cost is $m \times n$. If all of these are size $n$, then naive multiplication is $O(n^3)$.

```c
/* * end for i */ {
  /* * end for j */ {
    /* * end for k */ {
      c[i][j] += a[i][k] * b[k][j];
    }
    /* inner loop */ for (k = 1; k < n; k++)
      c[i][j] = 0.0;
  }
  /* loop on columns */ for (j = 1; j < n; j++)
    p[j] = 0.0;
}
/* loop on rows */ for (i = 1; i < n; i++)
  A[i] = B[i] * C;
```

Consider $A^{m \times d} \cdot B^{d \times u} = C^{m \times u}$ and count floating point multiplications.
And continue sparsifying the algorithm over the outer loops as well.

Multiply cost is \( m \times d \times \# \text{nonzeros} \), with an added cost from the sparse matrix data structure.

```c
/* * end for j */ {
    /* * end for i */ {
        /* * end for k */ {
            c[i][j] += a[i][k] * b[k][j];
        }
        for each nonzero pair a[i][k] and b[k][j]
            c[i][j] = 0.0;
    }
    /* * loop on columns */ {
        for (j = 0; j < p; j++) { // triple loop
            for (i = 0; i < m; i++) { // triple loop
                // triple loop
            }
        }
    }
}
```

1.6 Matrix Multiplication (2)