ceed at level 2, so **update[2]** also stores a pointer to the node with value 2. At level 1, we proceed to the node storing value 10. This is as far as we can go at level 1, so **update[1]** stores a pointer to the node with value 10. Finally, at level 0 we end up at the node with value 20. At this point, we can add in the new node with value 30. For each value **i**, the new node's **forward[i]** pointer is set to be **update[i]->forward[i]**, and the nodes stored in **update[i]** for indices 0 through 2 have their **forward[i]** pointers changed to point to the new node. This "splices" the new node into the Skip List at all levels.

The **remove** function is left as an exercise. It is similar to inserting in that the **update** array is built as part of searching for the record to be deleted; then those nodes specified by the update array have their forward pointers adjusted to point around the node being deleted.

A newly inserted node could have a high level generated by **randomLevel**, or a low level. It is possible that many nodes in the Skip List could have many pointers, leading to unnecessary insert cost and yielding poor (i.e., $\Theta(n)$) performance during search, since not many nodes will be skipped. Conversely, too many nodes could have a low level. In the worst case, all nodes could be at level 0, equivalent to a regular linked list. If so, search will again require $\Theta(n)$ time. However, the probability that performance will be poor is quite low. There is only once chance in 1024 that ten nodes in a row will be at level 0. The motto of probabilistic data structures such as the Skip List is "Don't worry, be happy." We simply accept the results of **randomLevel** and expect that probability will eventually work in our favor. The advantage of this approach is that the algorithms are simple, while requiring only $\Theta(\log n)$ time for all operations in the average case.

In practice, the Skip List will probably have better performance than a BST. The BST can have bad performance caused by the order in which data are inserted. For example, if n nodes are inserted into a BST in ascending order of their key value, then the BST will look like a linked list with the deepest node at depth $n - 1$. The Skip List's performance does not depend on the order in which values are inserted into the list. As the number of nodes in the Skip List increases, the probability of encountering the worst case decreases geometrically. Thus, the Skip List illustrates a tension between the theoretical worst case (in this case, $\Theta(n)$ for a Skip List operation), and a rapidly increasing probability of average-case performance of $\Theta(\log n)$, that characterizes probabilistic data structures.

16.4 Numerical Algorithms

Examples of problems:

- Raise a number to a power.
- Find common factors for two numbers.
- Tell whether a number is prime.
- Generate a random integer.
- Multiply two integers.

These operations use all the digits, and cannot use floating point approximation. For large numbers, cannot rely on hardware (constant time) operations. Measure input size by number of binary digits. Multiply, divide become expensive.

Analysis problem: Cost may depend on properties of the number other than size. It is easy to check an even number for primeness.

If you consider the cost over all k-bit inputs, cost grows with k. Features:

- Arithmetical operations are not cheap.
- There is only one instance of value n.
- There are 2^k instances of length k or less.
- The size (length) of value n is $\log n$.
- The cost may decrease when n increases in value, but generally increases when n increases in size (length).

16.4.1 Exponentiation

How do we compute m^n? We could multiply $n-1$ times. Can we do better? Approaches to divide and conquer:

- Relate m^n to k^n for $k < m$.
- Relate m^n to m^k for $k < n$.

If n is even, then $m^n = m^{n/2}m^{n/2}$. If n is odd, then $m^n = m^{\lfloor n/2 \rfloor}m^{\lfloor n/2 \rfloor}m$.

```
Power(base, exp) {
  if exp = 0 return 1;
  half = Power(base, exp/2);
  half = half * half;
  if (odd(exp)) then half = half * base;
  return half;
}
```

Analysis of Power:

$$f(n) = \begin{cases} 0 & n = 1 \\ f(\lfloor n/2 \rfloor) + 1 + n \bmod 2 & n > 1 \end{cases}$$

Solution:

$$f(n) = \lfloor \log n \rfloor + \beta(n) - 1$$

where β is the number of 1's in the binary representation of n.

How does this cost compare with the problem size? Is this the best possible? What if $n = 15$? What if n stays the same but m changes over many runs? In general, finding the best set of multiplications is expensive (probably exponential).

16.4.2 Largest Common Factor

The largest common factor of two numbers is the largest integer that divides both evenly. Observation: If k divides n and m, then k divides $n - m$. So, $f(n, m) = f(n - m, n) = f(m, n - m) = f(m, n)$. Observation: There exists k and l such that

$$n = km + l \text{ where } m > l \geq 0.$$

$$n = \lfloor n/m \rfloor m + n \bmod m.$$

So, $f(n, m) = f(m, l) = f(m, n \bmod m)$.

$$f(n, m) = \begin{cases} n & m = 0 \\ f(m, n \bmod m) & m > 0 \end{cases}$$

```
int LCF(int n, int m) {
  if (m == 0) return n;
  return LCF(m, n % m);
}
```

How big is $n \bmod m$ relative to n?

$$\begin{aligned} n \geq m &\Rightarrow n/m \geq 1 \\ &\Rightarrow 2\lfloor n/m \rfloor > n/m \\ &\Rightarrow m\lfloor n/m \rfloor > n/2 \\ &\Rightarrow n - n/2 > n - m\lfloor n/m \rfloor = n \bmod m \\ &\Rightarrow n/2 > n \bmod m \end{aligned}$$

The first argument must be halved in no more than 2 iterations. Total cost:

16.4.3 Matrix Multiplication

The standard algorithm for multiplying two $n \times n$ matrices requires $\Theta(n^3)$ time. It is possible to do better than this by rearranging and grouping the multiplications in various ways. One example of this is known as Strassen's matrix multiplication algorithm. Assume that n is a power of two. In the following, *A* and *B* are $n \times n$ arrays, while A_{ij} and B_{ij} refer to arrays of size $n/2 \times n/2$. Strassen's algorithm is to multiply the subarrays together in a particular order, as expressed by the following equation:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\ s_6 + s_7 & s_2 - s_3 + s_5 - s_7 \end{bmatrix}.$$

In other words, the result of the multiplication for an $n \times n$ array is obtained by a series of matrix multiplications and additions for $n/2 \times n/2$ arrays. Multiplications between subarrays also use Strassen's algorithm, and the addition of two subarrays requires $\Theta(n^2)$ time. The subfactors are defined as follows:

$$\begin{aligned} s_1 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\ s_2 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ s_3 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \\ s_4 &= (A_{11} + A_{12}) \cdot B_{22} \\ s_5 &= A_{11} \cdot (B_{12} - B_{22}) \\ s_6 &= A_{22} \cdot (B_{21} - B_{11}) \\ s_7 &= (A_{21} + A_{22}) \cdot B_{11}. \end{aligned}$$

1. Show that Strassen's algorithm is correct.
2. How many multiplications of subarrays and how many additions are required by Strassen's algorithm? How many would be required by normal matrix multiplication if it were defined in terms of subarrays in the same manner? Show the recurrence relations for both Strassen's algorithm and the normal matrix multiplication algorithm.
3. Derive the closed-form solution for the recurrence relation you gave for Strassen's algorithm (use Theorem 14.1).
4. Give your opinion on the practicality of Strassen's algorithm.

Given: $n \times n$ matrices A and B. Compute: $C = A \times B$.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Straightforward algorithm requires $\Theta(n^3)$ multiplications and additions.

Lower bound for any matrix multiplication algorithm: $\Omega(n^2)$.

Another Approach — Compute:

$$\begin{aligned} m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\ m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\ m_4 &= (a_{11} + a_{12})b_{22} \\ m_5 &= a_{11}(b_{12} - b_{22}) \\ m_6 &= a_{22}(b_{21} - b_{11}) \\ m_7 &= (a_{21} + a_{22})b_{11} \end{aligned}$$

Then:

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6 \\ c_{12} &= m_4 + m_5 \\ c_{21} &= m_6 + m_7 \\ c_{22} &= m_2 - m_3 + m_5 - m_7 \end{aligned}$$

This requires 7 multiplications and 18 additions/subtractions.

Strassen's Algorithm (1) Trade more additions/subtractions for fewer multiplications in 2×2 case. (2) Divide and conquer. In the straightforward implementation, 2×2 case is:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

Requires 8 multiplications and 4 additions.

Divide and conquer step: Assume n is a power of 2. Express $C = A \times B$ in terms of $\frac{n}{2} \times \frac{n}{2}$ matrices. By Strassen's algorithm, this can be computed with 7 multiplications and 18 additions/subtractions of $n/2 \times n/2$ matrices.

Recurrence:

$$\begin{aligned} T(n) &= 7T(n/2) + 18(n/2)^2 \\ T(n) &= \Theta(n^{\log_2 7}) = \Theta(n^{2.81}). \end{aligned}$$

Current "fastest" algorithm is $\Theta(n^{2.376})$ Open question: Can matrix multiplication be done in $O(n^2)$ time?

16.4.4 Random Numbers

Which sequences are random?

- 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
- 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
- 2, 7, 1, 8, 2, 8, 1, 8, 2, ...

Meanings of "random":

- Cannot predict the next item: **unpredictable**.
- Series cannot be described more briefly than to reproduce it: **equidistribution**.

There is no such thing as a random number sequence, only "random enough" sequences. A sequence is **pseudorandom** if no future term can be predicted in polynomial time, given all past terms.

Most computer systems use a deterministic algorithm to select pseudorandom numbers. **Linear congruential method**: Pick a **seed** $r(1)$. Then,

$$r(i) = (r(i-1) \times b) \bmod t.$$

Resulting numbers must be in range: What happens if $r(i) = r(j)$? Must pick good values for b and t. t should be prime.

Examples:

$r(i) = 6r(i-1) \bmod 13 =$
..., 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1, ...
$r(i) = 7r(i-1) \bmod 13 =$
..., 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1, ...
$r(i) = 5r(i-1) \bmod 13 =$
..., 1, 5, 12, 8, 1, ...
..., 2, 10, 11, 3, 2, ...
..., 4, 7, 9, 6, 4, ...
..., 0, 0, ...

Suggested generator:

$$r(i) = 16807r(i-1) \bmod 2^{31} - 1.$$

16.4.5 Fast Fourier Transform

An example of a useful reduction is multiplication through the use of logarithms. Multiplication is considerably more difficult than addition, since the cost to multiply two n-bit numbers directly is $O(n^2)$, while addition of two n-bit numbers is $O(n)$. Recall from Section 2.3 that one property of logarithms is

$$\log nm = \log n + \log m.$$

Thus, if taking logarithms and anti-logarithms were cheap, then we could reduce multiplication to addition by taking the log of the two operands, adding, and then taking the anti-log of the sum.

Under normal circumstances, taking logarithms and anti-logarithms is expensive, and so this reduction would not be considered practical. However, this reduction is precisely the basis for the slide rule. The slide rule uses a logarithmic scale to measure the lengths of two numbers, in effect doing the conversion to logarithms automatically. These two lengths are then added together, and the inverse logarithm of the sum is read off another logarithmic scale. The part normally considered expensive (taking logarithms and anti-logarithms) is cheap since it is a physical part of the slide rule. Thus, the entire multiplication process can be done cheaply via a reduction to addition.

Compared to addition, multiplication is hard. In the physical world, addition is merely concatenating two lengths. Observation:

$$\log nm = \log n + \log m.$$

Therefore,

$$nm = \text{ antilog}(\log n + \log m).$$

What if taking logs and antilogs were easy? The slide rule does exactly this! It is essentially two rulers in log scale. Slide the scales to add the lengths of the two numbers (in log form). The third scale shows the value for the total length.

Now lets consider multiplying polynomials. A vector $\mathbf{a}$ of n values can uniquely represent a polynomial of degree $n-1$

$$P_{\mathbf{a}}(x) = \sum_{i=0}^{n-1} \mathbf{a}_i x^i.$$

Alternatively, a polynomial can be uniquely represented by a list of its values at n distinct points. Finding the value for a polynomial at a given point is called

evaluation. Finding the coefficients for the polynomial given the values at n points is called **interpolation**.

To multiply two $n-1$-degree polynomials A and B normally takes $\Theta(n^2)$ coefficient multiplications. However, if we evaluate both polynomials (at the same points), we can simply multiply the corresponding pairs of values to get the corresponding values for polynomial AB. Process:

- Evaluate polynomials A and B at enough points.
- Pairwise multiplications of resulting values.
- Interpolation of resulting values.

This can be faster than $\Theta(n^2)$ IF a fast way can be found to do evaluation/interpolation of $2n-1$ points. Normally this takes $\Theta(n^2)$ time. (Why?)

Example 16.2 Polynomial A: x^2+1. Polynomial B: $2x^2-x+1$. Polynomial AB: $2x^4-x^3+3x^2-x+1$.

Note that evaluating a polynomial at 0 is easy. If we evaluate at 1 and -1, we can share a lot of the work between the two evaluations. Can we find enough such points to make the process cheap?

$$\begin{aligned} AB(-1) &= (2)(4) = 8 \\ AB(0) &= (1)(1) = 1 \\ AB(1) &= (2)(2) = 4 \end{aligned}$$

But: We need 5 points to nail down Polynomial AB. And, we also need to interpolate the 5 values to get the coefficients back.

Observation: In general, we can write $P_a(x) = E_a(x) + O_a(x)$ where E_a is the even powers and O_a is the odd powers. So,

$$P_a(x) = \sum_{i=0}^{n/2-1} a_{2i}x^{2i} + \sum_{i=0}^{n/2-1} a_{2i+1}x^{2i+1}$$

The significance is that when evaluating the pair of values x and $-x$, we get

$$\begin{aligned} E_a(x) + O_a(x) &= E_a(x) - O_a(-x) \\ O_a(x) &= -O_a(-x) \end{aligned}$$

Thus, we only need to compute the E's and O's once instead of twice to get both evaluations.

The key to fast polynomial multiplication is finding the right points to use for evaluation/interpolation to make the process efficient. Complex number z is a **primitive nth root of unity** if

1. $z^n = 1$ and

2. $z^k \neq 1$ for $0 < k < n$.

$z^0, z^1, ..., z^{n-1}$ are the **nth roots of unity**. Example: For $n = 4$, $z = i$ or $z = -i$. Identity: $e^{i\pi} = -1$.

In general, $z^j = e^{2\pi ij/n} = -1^{2j/n}$. Significance: We can find as many points on the circle as we need.

Define an $n \times n$ matrix A_z with row i and column j as

$$A_z = (z^{ij}).$$

Example: $n = 4$, $z = i$:

$$A_z = \begin{matrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{matrix}$$

Let $a = [a_0, a_1, ..., a_{n-1}]^T$ be a vector. We can evaluate the polynomial at the nth roots of unity:

$$F_z = A_z a = b.$$

$$b_i = \sum_{k=0}^{n-1} a_k z^{ik}.$$

For $n = 8$, $z = \sqrt{i}$. So,

$$A_z = \begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \sqrt{i} & i & i\sqrt{i} & -1 & -\sqrt{i} & -i & -i\sqrt{i} \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & i\sqrt{i} & -i & \sqrt{i} & -1 & -i\sqrt{i} & i & -\sqrt{i} \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\sqrt{i} & i & -i\sqrt{i} & -1 & \sqrt{i} & -i & i\sqrt{i} \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & -i\sqrt{i} & -i & -\sqrt{i} & -1 & i\sqrt{i} & i & \sqrt{i} \end{matrix}$$

We still have two problems: We need to be able to do this fast. Its still n^2 multiplies to evaluate. If we multiply the two sets of evaluations (cheap), we still need to be able to reverse the process (interpolate).

The interpolation step is nearly identical to the evaluation step.

$$F_z^{-1} = A_z^{-1} b' = a'.$$

What is A_z^{-1}? This turns out to be simple to compute.

$$A_z^{-1} = \frac{1}{n} A_{1/z}.$$

In other words, do the same computation as before but substitute $1/z$ for z (and multiply by $1/n$ at the end). So, if we can do one fast, we can do the other fast.

An efficient divide and conquer algorithm exists to perform both the evaluation and the interpolation in $\Theta(n \log n)$ time. This is called the **Discrete Fourier Transform** (DFT). It is a recursive function that decomposes the matrix multiplications, taking advantage of the symmetries made available by doing evaluation at the nth roots of unity.

Polynomial multiplication of A and B:

- Represent an $n-1$-degree polynomial as $2n-1$ coefficients:

$$[a_0, a_1, ..., a_{n-1}, 0, ..., 0]$$

- Perform DFT on representations for A and B
- Pairwise multiply results to get $2n-1$ values.
- Perform inverse DFT on result to get $2n-1$ degree polynomial AB.

```
Fourier_Transform(double *Polynomial, int n) {
  // Compute the Fourier transform of Polynomial
  // with degree n. Polynomial is a list of
  // coefficients indexed from 0 to n-1. n is
  // assumed to be a power of 2.
  double Even[n/2], Odd[n/2], List1[n/2], List2[n/2];

  if (n==1) return Polynomial[0];

  for (j=0; j<=n/2-1; j++) {
    Even[j] = Polynomial[2j];
    Odd[j] = Polynomial[2j+1];
  }
  List1 = Fourier_Transform(Even, n/2);
```

```
  List2 = Fourier_Transform(Odd, n/2);
  for (j=0; j<=n-1, J++) {
    Imaginary z = pow(E, 2*i*PI*j/n);
    k = j % (n/2);
    Polynomial[j] = List1[k] + z*List2[k];
  }
  return Polynomial;
}
```

This just does the transform on one of the two polynomials. The full process is:

1. Transform each polynomial.
2. Multiply the resulting values ($O(n)$ multiplies).
3. Do the inverse transformation on the result.

16.5 Further Reading

For further information on Skip Lists, see "Skip Lists: A Probabilistic Alternative to Balanced Trees" by William Pugh [Pug90].

16.6 Exercises

16.1 Solve Towers of Hanoi using a dynamic programming algorithm.

16.2 There are six permutations of the lines

```
for (int k=0; k<G.n(); k++)
  for (int i=0; i<G.n(); i++)
    for (int j=0; j<G.n(); j++)
```

in floyd's algorithm. Which ones give a correct algorithm?

16.3 Show the result of running Floyd's all-pairs shortest-paths algorithm on the graph of Figure 11.25.

16.4 The implementation for Floyd's algorithm given in Section 16.2.2 is inefficient for adjacency lists because the edges are visited in a bad order when initializing array `D`. What is the cost of of this initialization step for the adjacency list? How can this initialization step be revised so that it costs $\Theta(|\mathbf{V}|^2)$ in the worst case?

16.5 State the greatest possible lower bound that you can for the all-pairs shortest-paths problem, and justify your answer.

16.6 Show the Skip List that results from inserting the following values. Draw the Skip List after each insert. With each value, assume the depth of its corresponding node is as given in the list.