16

Patterns of Algorithms

This chapter presents several fundamental topics related to the theory of algorithms. Included are dynamic programming (Section 16.1), randomized algorithms (Section 16.2), and the concept of a transform (Section 16.3.5). Each of these can be viewed as an example of an "algorithmic pattern" that is commonly used for a wide variety of applications. In addition, Section 16.3 presents a number of numerical algorithms. Section 16.2 on randomized algorithms includes the Skip List (Section 16.2.2). The Skip List is a probabilistic data structure that can be used to implement the dictionary ADT. The Skip List is no more complicated than the BST. Yet it often outperforms the BST because the Skip List's efficiency is not tied to the values or insertion order of the dataset being stored.

16.1 Dynamic Programming

Consider again the recursive function for computing the nth Fibonacci number.

```
/** Recursively generate and return the n'th Fibonacci
    number */
static long fibr(int n) {
  // fibr(91) is the largest value that fits in a long
  assert (n > 0) && (n <= 91) : "n out of range";
  if ((n == 1) || (n == 2)) return 1;     // Base case
  return fibr(n-1) + fibr(n-2);      // Recursive call
}
```

The cost of this algorithm (in terms of function calls) is the size of the nth Fibonacci number itself, which our analysis of Section 14.2 showed to be exponential (approximately $n^{1.62}$). Why is this so expensive? Primarily because two recursive calls are made by the function, and the work that they do is largely redundant. That is, each of the two calls is recomputing most of the series, as is each sub-call, and so on. Thus, the smaller values of the function are being recomputed a huge number of times. If we could eliminate this redundancy, the cost would be greatly reduced.

The approach that we will use can also improve any algorithm that spends most of its time recomputing common subproblems.

One way to accomplish this goal is to keep a table of values, and first check the table to see if the computation can be avoided. Here is a straightforward example of doing so.

```
int fibrt(int n) {
  // Assume Values has at least n slots, and all
  // slots are initialized to 0
  if (n <= 2) return 1;             // Base case
  if (Values[n] == 0)
    Values[n] = fibrt(n-1) + fibrt(n-2);
  return Values[n];
}
```

This version of the algorithm will not compute a value more than once, so its cost should be linear. Of course, we didn't actually need to use a table storing all of the values, since future computations do not need access to all prior subproblems. Instead, we could build the value by working from 0 and 1 up to n rather than backwards from n down to 0 and 1. Going up from the bottom we only need to store the previous two values of the function, as is done by our iterative version.

```
/** Iteratively generate and return the n'th Fibonacci
    number */
static long fibi(int n) {
  // fibr(91) is the largest value that fits in a long
  assert (n > 0) && (n <= 91) : "n out of range";
  long curr, prev, past;
  if ((n == 1) || (n == 2)) return 1;
  curr = prev = 1;      // curr holds current Fib value
  for (int i=3; i<=n; i++) { // Compute next value
    past = prev;             // past holds fibi(i-2)
    prev = curr;             // prev holds fibi(i-1)
    curr = past + prev;      // curr now holds fibi(i)
  }
  return curr;
}
```

Recomputing of subproblems comes up in many algorithms. It is not so common that we can store only a few prior results as we did for **`fibi`**. Thus, there are many times where storing a complete table of subresults will be useful.

This approach to designing an algorithm that works by storing a table of results for subproblems is called dynamic programming. The name is somewhat arcane, because it doesn't bear much obvious similarity to the process that is taking place when storing subproblems in a table. However, it comes originally from the field of dynamic control systems, which got its start before what we think of as computer programming. The act of storing precomputed values in a table for later reuse is referred to as "programming" in that field.

Dynamic programming is a powerful alternative to the standard principle of divide and conquer. In divide and conquer, a problem is split into subproblems, the subproblems are solved (independently), and then recombined into a solution for the problem being solved. Dynamic programming is appropriate whenever (1) subproblems are solved repeatedly, and (2) we can find a suitable way of doing the necessary bookkeeping. Dynamic programming algorithms are usually not implemented by simply using a table to store subproblems for recursive calls (i.e., going backwards as is done by **`fibrt`**). Instead, such algorithms are typically implemented by building the table of subproblems from the bottom up. Thus, **`fibi`** better represents the most common form of dynamic programming than does **`fibrt`**, even though it doesn't use the complete table.

16.1.1 The Knapsack Problem

We will next consider a problem that appears with many variations in a variety of commercial settings. Many businesses need to package items with the greatest efficiency. One way to describe this basic idea is in terms of packing items into a knapsack, and so we will refer to this as the Knapsack Problem. We will first define a particular formulation of the knapsack problem, and then we will discuss an algorithm to solve it based on dynamic programming. We will see other versions of the knapsack problem in the exercises and in Chapter 17.

Assume that we have a knapsack with a certain amount of space that we will define using integer value K. We also have n items each with a certain size such that that item i has integer size k_i. The problem is to find a subset of the n items whose sizes exactly sum to K, if one exists. For example, if our knapsack has capacity $K = 5$ and the two items are of size $k_1 = 2$ and $k_2 = 4$, then no such subset exists. But if we add a third item of size $k_3 = 1$, then we can fill the knapsack exactly with the second and third items. We can define the problem more formally as: Find $S \subset \{1, 2, ..., n\}$ such that

$$\sum_{i \in S} k_i = K.$$

Example 16.1 Assume that we are given a knapsack of size $K = 163$ and 10 items of sizes 4, 9, 15, 19, 27, 44, 54, 68, 73, 101. Can we find a subset of the items that exactly fills the knapsack? You should take a few minutes and try to do this before reading on and looking at the answer.

One solution to the problem is: 19, 27, 44, 73.

Example 16.2 Having solved the previous example for knapsack of size 163, how hard is it now to solve for a knapsack of size 164?

Unfortunately, knowing the answer for 163 is of almost no use at all when solving for 164. One solution is: 9, 54, 101.

If you tried solving these examples, you probably found yourself doing a lot of trial-and-error and a lot of backtracking. To come up with an algorithm, we want an organized way to go through the possible subsets. Is there a way to make the problem smaller, so that we can apply divide and conquer? We essentially have two parts to the input: The knapsack size K and the n items. It probably will not do us much good to try and break the knapsack into pieces and solve the sub-pieces (since we already saw that knowing the answer for a knapsack of size 163 did nothing to help us solve the problem for a knapsack of size 164).

So, what can we say about solving the problem with or without the nth item? This seems to lead to a way to break down the problem. If the nth item is not needed for a solution (that is, if we can solve the problem with the first $n-1$ items) then we can also solve the problem when the nth item is available (we just ignore it). On the other hand, if we do include the nth item as a member of the solution subset, then we now would need to solve the problem with the first $n-1$ items and a knapsack of size $K - k_n$ (since the nth item is taking up k_n space in the knapsack).

To organize this process, we can define the problem in terms of two parameters: the knapsack size K and the number of items n. Denote a given instance of the problem as $P(n, K)$. Now we can say that $P(n, K)$ has a solution if and only if there exists a solution for either $P(n-1, K)$ or $P(n-1, K-k_n)$. That is, we can solve $P(n, K)$ only if we can solve one of the sub problems where we use or do not use the nth item. Of course, the ordering of the items is arbitrary. We just need to give them some order to keep things straight.

Continuing this idea, to solve any subproblem of size $n-1$, we need only to solve two subproblems of size $n-2$. And so on, until we are down to only one item that either fills the knapsack or not. This naturally leads to a cost expressed by the recurrence relation $T(n) = 2T(n-1) + c = \Theta(2^n)$. That can be pretty expensive!

But... we should quickly realize that there are only $n(K+1)$ subproblems to solve! Clearly, there is the possibility that many subproblems are being solved repeatedly. This is a natural opportunity to apply dynamic programming. We simply build an array of size $n \times K + 1$ to contain the solutions for all subproblems $P(i, k), 1 \leq i \leq n, 0 \leq k \leq K$.

There are two approaches to actually solving the problem. One is to start with our problem of size $P(n, K)$ and make recursive calls to solve the subproblems, each time checking the array to see if a subproblem has been solved, and filling in the corresponding cell in the array whenever we get a new subproblem solution. The other is to start filling the array for row 1 (which indicates a successful solution

only for a knapsack of size k_1). We then fill in the succeeding rows from $i = 2$ to n, left to right, as follows.

if $P(n-1, K)$ has a solution,
  then $P(n, K)$ has a solution
  else if $P(n-1, K-k_n)$ has a solution
    then $P(n, K)$ has a solution
    else $P(n, K)$ has no solution.

In other words, a new slot in the array gets its solution by looking at two slots in the preceding row. Since filling each slot in the array takes constant time, the total cost of the algorithm is $\Theta(nK)$.

Example 16.3 Solve the Knapsack Problem for $K = 10$ and five items with sizes 9, 2, 7, 4, 1. We do this by building the following array.

	0	1	2	3	4	5	6	7	8	9	10
$k_1 = 9$	O	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	I	$-$
$k_2 = 2$	O	$-$	I	$-$	$-$	$-$	$-$	$-$	$-$	O	$-$
$k_3 = 7$	O	$-$	O	$-$	$-$	$-$	$-$	I	$-$	I/O	$-$
$k_4 = 4$	O	$-$	O	$-$	I	$-$	I	O	$-$	O	$-$
$k_5 = 1$	O	I	O	I	O	I	O	I/O	I	O	I

Key:
-: No solution for $P(i, k)$.
O: Solution(s) for $P(i, k)$ with i omitted.
I: Solution(s) for $P(i, k)$ with i included.
I/O: Solutions for $P(i, k)$ with i included AND omitted.

For example, $P(3, 9)$ stores value I/O. It contains O because $P(2, 9)$ has a solution. It contains I because $P(2, 2) = P(2, 9-7)$ has a solution. Since $P(5, 10)$ is marked with an I, it has a solution. We can determine what that solution actually is by recognizing that it includes the 5th item (of size 1), which then leads us to look at the solution for $P(4, 9)$. This in turn has a solution that omits the 4th item, leading us to $P(3, 9)$. At this point, we can either use the third item or not. We can find a solution by taking one branch. We can find all solutions by following all branches when there is a choice.

16.1.2 All-Pairs Shortest Paths

We next consider the problem of finding the shortest distance between all pairs of vertices in the graph, called the **all-pairs shortest-paths** problem. To be precise, for every $u, v \in \mathbf{V}$, calculate d(u, v).

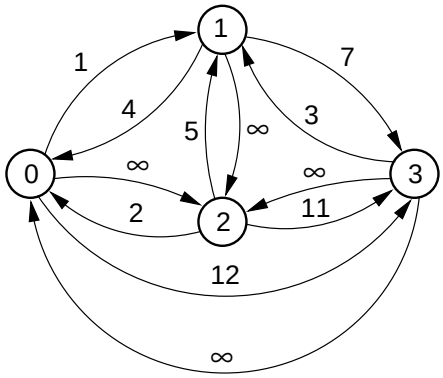

Figure 16.1 An example of k-paths in Floyd's algorithm. Path 1, 3 is a 0-path by definition. Path 3, 0, 2 is not a 0-path, but it is a 1-path (as well as a 2-path, a 3-path, and a 4-path) because the largest intermediate vertex is 0. Path 1, 3, 2 is a 4-path, but not a 3-path because the intermediate vertex is 3. All paths in this graph are 4-paths.

One solution is to run Dijkstra's algorithm for finding the single-source shortest path (see Section 11.4.1) $|\mathbf{V}|$ times, each time computing the shortest path from a different start vertex. If $\mathbf{G}$ is sparse (that is, $|\mathbf{E}| = \Theta(|\mathbf{V}|)$) then this is a good solution, because the total cost will be $\Theta(|\mathbf{V}|^2 + |\mathbf{V}||\mathbf{E}| \log |\mathbf{V}|) = \Theta(|\mathbf{V}|^2 \log |\mathbf{V}|)$ for the version of Dijkstra's algorithm based on priority queues. For a dense graph, the priority queue version of Dijkstra's algorithm yields a cost of $\Theta(|\mathbf{V}|^3 \log |\mathbf{V}|)$, but the version using `MinVertex` yields a cost of $\Theta(|\mathbf{V}|^3)$.

Another solution that limits processing time to $\Theta(|\mathbf{V}|^3)$ regardless of the number of edges is known as Floyd's algorithm. It is an example of dynamic programming. The chief problem with solving this problem is organizing the search process so that we do not repeatedly solve the same subproblems. We will do this organization through the use of the k-path. Define a ***k*-path** from vertex *v* to vertex *u* to be any path whose intermediate vertices (aside from *v* and *u*) all have indices less than k. A 0-path is defined to be a direct edge from *v* to *u*. Figure 16.1 illustrates the concept of k-paths.

Define $\mathrm{D}_k(v, u)$ to be the length of the shortest k-path from vertex *v* to vertex *u*. Assume that we already know the shortest k-path from *v* to *u*. The shortest $(k+1)$-path either goes through vertex k or it does not. If it does go through k, then the best path is the best k-path from *v* to k followed by the best k-path from k to *u*. Otherwise, we should keep the best k-path seen before. Floyd's algorithm simply checks all of the possibilities in a triple loop. Here is the implementation for Floyd's algorithm. At the end of the algorithm, array `D` stores the all-pairs shortest distances.

```
/** Compute all-pairs shortest paths */
static void Floyd(Graph G, int[][] D) {
  for (int i=0; i<G.n(); i++) // Initialize D with weights
    for (int j=0; j<G.n(); j++)
      if (G.weight(i, j) != 0) D[i][j] = G.weight(i, j);
  for (int k=0; k<G.n(); k++) // Compute all k paths
    for (int i=0; i<G.n(); i++)
      for (int j=0; j<G.n(); j++)
        if ((D[i][k] != Integer.MAX_VALUE) &&
            (D[k][j] != Integer.MAX_VALUE) &&
            (D[i][j] > (D[i][k] + D[k][j])))
          D[i][j] = D[i][k] + D[k][j];
}
```

Clearly this algorithm requires $\Theta(|\mathbf{V}|^3)$ running time, and it is the best choice for dense graphs because it is (relatively) fast and easy to implement.

16.2 Randomized Algorithms

In this section, we will consider how introducing randomness into our algorithms might speed things up, although perhaps at the expense of accuracy. But often we can reduce the possibility for error to be as low as we like, while still speeding up the algorithm.

16.2.1 Randomized algorithms for finding large values

In Section 15.1 we determined that the lower bound cost of finding the maximum value in an unsorted list is $\Omega(n)$. This is the least time needed to be certain that we have found the maximum value. But what if we are willing to relax our requirement for certainty? The first question is: What do we mean by this? There are many aspects to "certainty" and we might relax the requirement in various ways.

There are several possible guarantees that we might require from an algorithm that produces X as the maximum value, when the true maximum is Y. So far we have assumed that we require X to equal Y. This is known as an exact or deterministic algorithm to solve the problem. We could relax this and require only that X's rank is "close to" Y's rank (perhaps within a fixed distance or percentage). This is known as an approximation algorithm. We could require that X is "usually" Y. This is known as a probabilistic algorithm. Finally, we could require only that X's rank is "usually" "close" to Y's rank. This is known as a heuristic algorithm.

There are also different ways that we might choose to sacrifice reliability for speed. These types of algorithms also have names.

1. **Las Vegas Algorithms**: We always find the maximum value, and "usually" we find it fast. Such algorithms have a guaranteed result, but do not guarantee fast running time.

2. **Monte Carlo Algorithms**: We find the maximum value fast, or we don't get an answer at all (but fast). While such algorithms have good running time, their result is not guaranteed.

Here is an example of an algorithm for finding a large value that gives up its guarantee of getting the best value in exchange for an improved running time. This is an example of a **probabilistic** algorithm, since it includes steps that are affected by **random** events. Choose m elements at random, and pick the best one of those as the answer. For large n, if $m \approx \log n$, the answer is pretty good. The cost is $m - 1$ compares (since we must find the maximum of m values). But we don't know for sure what we will get. However, we can estimate that the rank will be about $\frac{mn}{m+1}$. For example, if $n = 1,000,000$ and $m = \log n = 20$, then we expect that the largest of the 20 randomly selected values be among the top 5% of the n values.

Next, consider a slightly different problem where the goal is to pick a number in the upper half of n values. We would pick the maximum from among the first $\frac{n+1}{2}$ values for a cost of $n/2$ comparisons. Can we do better than this? Not if we want to guarantee getting the correct answer. But if we are willing to accept near certainty instead of absolute certainty, we can gain a lot in terms of speed.

As an alternative, consider this probabilistic algorithm. Pick 2 numbers and choose the greater. This will be in the upper half with probability 3/4 (since it is not in the upper half only when both numbers we choose happen to be in the lower half). Is a probability of 3/4 not good enough? Then we simply pick more numbers! For k numbers, the greatest is in upper half with probability $1 - \frac{1}{2^k}$, regardless of the number n that we pick from, so long as n is much larger than k (otherwise the chances might become even better). If we pick ten numbers, then the chance of failure is only one in $2^{10} = 1024$. What if we really want to be sure, because lives depend on drawing a number from the upper half? If we pick 30 numbers, we can fail only one time in a billion. If we pick enough numbers, then the chance of picking a small number is less than the chance of the power failing during the computation. Picking 100 numbers means that we can fail only one time in 10^{100} which is less chance than any disaster that you can imagine disrupting the process.

16.2.2 Skip Lists

This section presents a probabilistic search structure called the Skip List. Like BSTs, Skip Lists are designed to overcome a basic limitation of array-based and linked lists: Either search or update operations require linear time. The Skip List is an example of a **probabilistic data structure**, because it makes some of its decisions at random.

Skip Lists provide an alternative to the BST and related tree structures. The primary problem with the BST is that it may easily become unbalanced. The 2-3 tree of Chapter 10 is guaranteed to remain balanced regardless of the order in which data

values are inserted, but it is rather complicated to implement. Chapter 13 presents the AVL tree and the splay tree, which are also guaranteed to provide good performance, but at the cost of added complexity as compared to the BST. The Skip List is easier to implement than known balanced tree structures. The Skip List is not guaranteed to provide good performance (where good performance is defined as $\Theta(\log n)$ search, insertion, and deletion time), but it will provide good performance with extremely high probability (unlike the BST which has a good chance of performing poorly). As such it represents a good compromise between difficulty of implementation and performance.

Figure 16.2 illustrates the concept behind the Skip List. Figure 16.2(a) shows a simple linked list whose nodes are ordered by key value. To search a sorted linked list requires that we move down the list one node at a time, visiting $\Theta(n)$ nodes in the average case. What if we add a pointer to every other node that lets us skip alternating nodes, as shown in Figure 16.2(b)? Define nodes with a single pointer as level 0 Skip List nodes, and nodes with two pointers as level 1 Skip List nodes.

To search, follow the level 1 pointers until a value greater than the search key has been found, go back to the previous level 1 node, then revert to a level 0 pointer to travel one more node if necessary. This effectively cuts the work in half. We can continue adding pointers to selected nodes in this way — give a third pointer to every fourth node, give a fourth pointer to every eighth node, and so on — until we reach the ultimate of $\log n$ pointers in the first and middle nodes for a list of n nodes as illustrated in Figure 16.2(c). To search, start with the bottom row of pointers, going as far as possible and skipping many nodes at a time. Then, shift up to shorter and shorter steps as required. With this arrangement, the worst-case number of accesses is $\Theta(\log n)$.

We will store with each Skip List node an array named **`forward`** that stores the pointers as shown in Figure 16.2(c). Position **`forward[0]`** stores a level 0 pointer, **`forward[1]`** stores a level 1 pointer, and so on. The Skip List object includes data member **`level`** that stores the highest level for any node currently in the Skip List. The Skip List stores a header node named **`head`** with **`level`** pointers. The **`find`** function is shown in Figure 16.3.

Searching for a node with value 62 in the Skip List of Figure 16.2(c) begins at the header node. Follow the header node's pointer at **`level`**, which in this example is level 2. This points to the node with value 31. Because 31 is less than 62, we next try the pointer from **`forward[2]`** of 31's node to reach 69. Because 69 is greater than 62, we cannot go forward but must instead decrement the current level counter to 1. We next try to follow **`forward[1]`** of 31 to reach the node with value 58. Because 58 is smaller than 62, we follow 58's **`forward[1]`** pointer to 69. Because 69 is too big, follow 58's level 0 pointer to 62. Because 62 is not less than 62, we fall out of the **`while`** loop and move one step forward to the node with value 62.

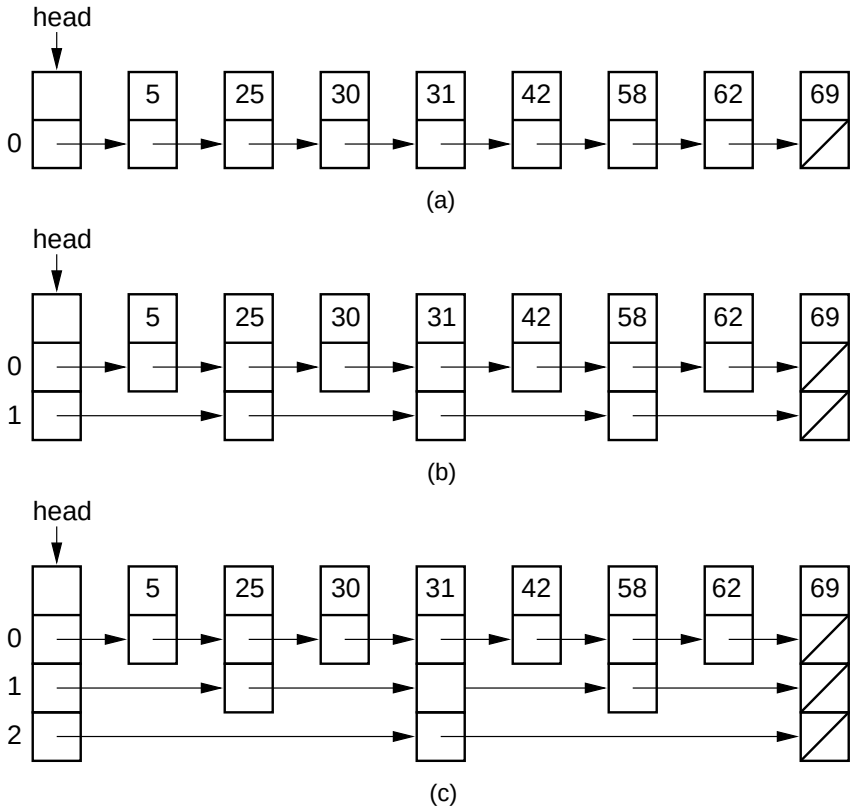

Figure 16.2 Illustration of the Skip List concept. (a) A simple linked list. (b) Augmenting the linked list with additional pointers at every other node. To find the node with key value 62, we visit the nodes with values 25, 31, 58, and 69, then we move from the node with key value 58 to the one with value 62. (c) The ideal Skip List, guaranteeing $\mathrm{O}(\log n)$ search time. To find the node with key value 62, we visit nodes in the order 31, 69, 58, then 69 again, and finally, 62.

```
/** Skiplist Search */
public E find(Key searchKey) {
  SkipNode<Key,E> x = head;           // Dummy header node
  for (int i=level; i>=0; i--)        // For each level...
    while ((x.forward[i] != null) && // go forward
           (searchKey.compareTo(x.forward[i].key()) > 0))
      x = x.forward[i];               // Go one last step
  x = x.forward[0];  // Move to actual record, if it exists
  if ((x != null) && (searchKey.compareTo(x.key()) == 0))
    return x.element();               // Got it
  else return null;                   // Its not there
}
```

Figure 16.3 Implementation for the Skip List **find** function.

```
/** Insert a record into the skiplist */
public void insert(Key k, E newValue) {
  int newLevel = randomLevel();  // New node's level
  if (newLevel > level)          // If new node is deeper
    AdjustHead(newLevel);        //   adjust the header
  // Track end of level
  SkipNode<Key,E>[] update =
         (SkipNode<Key,E>[])new SkipNode[level+1];
  SkipNode<Key,E> x = head;        // Start at header node
  for (int i=level; i>=0; i--) { // Find insert position
    while((x.forward[i] != null) &&
          (k.compareTo(x.forward[i].key()) > 0))
      x = x.forward[i];
    update[i] = x;               // Track end at level i
  }
  x = new SkipNode<Key,E>(k, newValue, newLevel);
  for (int i=0; i<=newLevel; i++) {      // Splice into list
    x.forward[i] = update[i].forward[i]; // Who x points to
    update[i].forward[i] = x;            // Who y points to
  }
  size++;                          // Increment dictionary size
}
```

Figure 16.4 Implementation for the Skip List **`Insert`** function.

The ideal Skip List of Figure 16.2(c) has been organized so that (if the first and last nodes are not counted) half of the nodes have only one pointer, one quarter have two, one eighth have three, and so on. The distances are equally spaced; in effect this is a "perfectly balanced" Skip List. Maintaining such balance would be expensive during the normal process of insertions and deletions. The key to Skip Lists is that we do not worry about any of this. Whenever inserting a node, we assign it a level (i.e., some number of pointers). The assignment is random, using a geometric distribution yielding a 50% probability that the node will have one pointer, a 25% probability that it will have two, and so on. The following function determines the level based on such a distribution:

```
/** Pick a level using a geometric distribution */
int randomLevel() {
  int lev;
  for (lev=0; DSutil.random(2) == 0; lev++); // Do nothing
  return lev;
}
```

Once the proper level for the node has been determined, the next step is to find where the node should be inserted and link it in as appropriate at all of its levels. Figure 16.4 shows an implementation for inserting a new value into the Skip List.

Figure 16.5 illustrates the Skip List insertion process. In this example, we begin by inserting a node with value 10 into an empty Skip List. Assume that **`randomLevel`** returns a value of 1 (i.e., the node is at level 1, with 2 pointers). Because the empty Skip List has no nodes, the level of the list (and thus the level

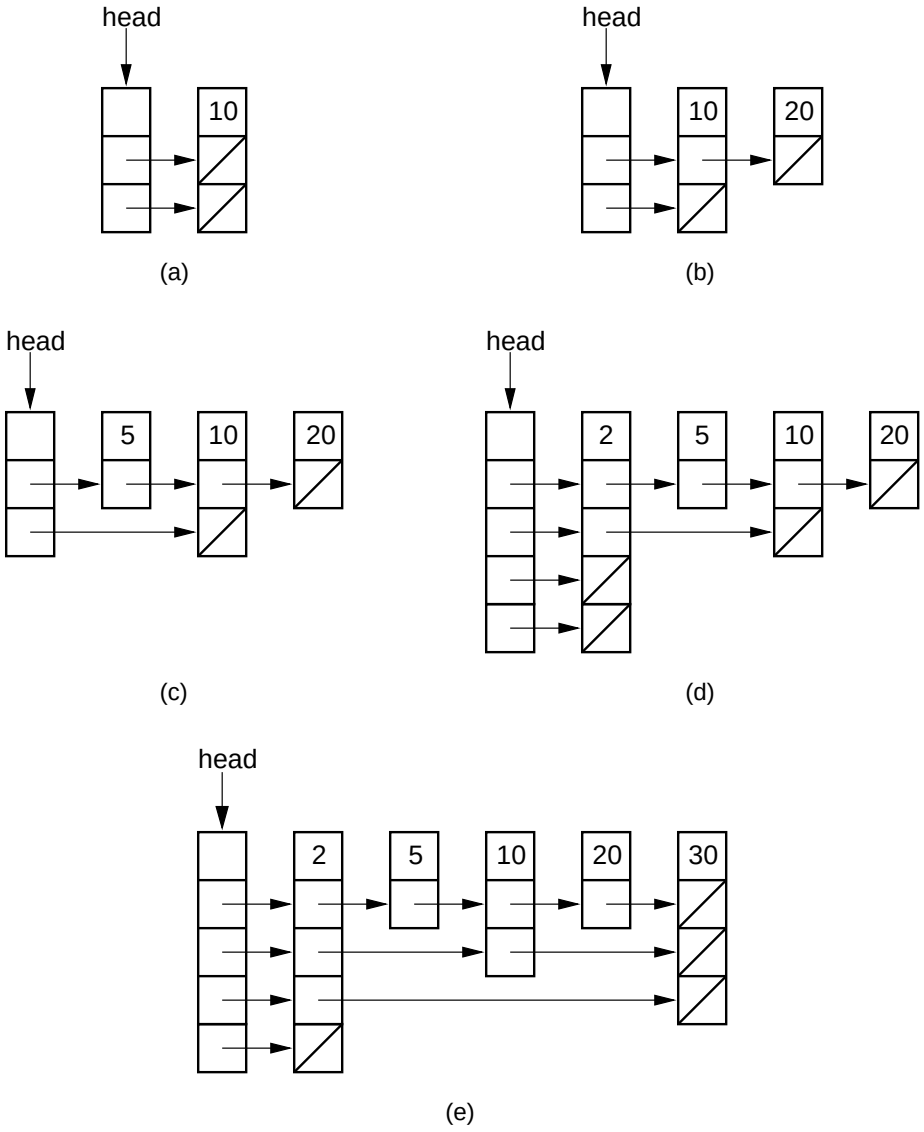

Figure 16.5 Illustration of Skip List insertion. (a) The Skip List after inserting initial value 10 at level 1. (b) The Skip List after inserting value 20 at level 0. (c) The Skip List after inserting value 5 at level 0. (d) The Skip List after inserting value 2 at level 3. (e) The final Skip List after inserting value 30 at level 2.

of the header node) must be set to 1. The new node is inserted, yielding the Skip List of Figure 16.5(a).

Next, insert the value 20. Assume this time that **randomLevel** returns 0. The search process goes to the node with value 10, and the new node is inserted after, as shown in Figure 16.5(b). The third node inserted has value 5, and again assume that **randomLevel** returns 0. This yields the Skip List of Figure 16.5.c.

The fourth node inserted has value 2, and assume that **randomLevel** returns 3. This means that the level of the Skip List must rise, causing the header node to gain an additional two (**null**) pointers. At this point, the new node is added to the front of the list, as shown in Figure 16.5(d).

Finally, insert a node with value 30 at level 2. This time, let us take a close look at what array **update** is used for. It stores the farthest node reached at each level during the search for the proper location of the new node. The search process begins in the header node at level 3 and proceeds to the node storing value 2. Because **forward[3]** for this node is **null**, we cannot go further at this level. Thus, **update[3]** stores a pointer to the node with value 2. Likewise, we cannot proceed 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 insertion 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, because 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 one 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.

16.3 Numerical Algorithms

This section presents a variety of algorithms related to mathematical computations on numbers. Examples are activities like multiplying two numbers or raising a number to a given power. In particular, we are concerned with situations where built-in integer or floating-point operations cannot be used because the values being operated on are too large. Similar concerns arise for operations on polynomials or matrices.

Since we cannot rely on the hardware to process the inputs in a single constant-time operation, we are concerned with how to most effectively implement the operation to minimize the time cost. This begs a question as to how we should apply our normal measures of asymptotic cost in terms of growth rates on input size. First, what is an instance of addition or multiplication? Each value of the operands yields a different problem instance. And what is the input size when multiplying two numbers? If we view the input size as two (since two numbers are input), then any non-constant-time algorithm has a growth rate that is infinitely high compared to the growth of the input. This makes no sense, especially in light of the fact that we know from grade school arithmetic that adding or multiplying numbers does seem to get more difficult as the value of the numbers involved increases. In fact, we know from standard grade school algorithms that the cost of standard addition is linear on the number of digits being added, and multiplication has cost $n \times m$ when multiplying an m-digit number by an n-digit number.

The number of digits for the operands does appear to be a key consideration when we are performing a numeric algorithm that is sensitive to input size. The number of digits is simply the log of the value, for a suitable base of the log. Thus, for the purpose of calculating asymptotic growth rates of algorithms, we will consider the "size" of an input value to be the log of that value. Given this view, there are a number of features that seem to relate such operations.

- Arithmetic operations on large values 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 of a particular algorithm can decrease when n increases in value (say when going from a value of $2^k - 1$ to 2^k to $2^k + 1$), but generally increases when n increases in length.

16.3.1 Exponentiation

We will start our examination of standard numerical algorithms by considering how to perform exponentiation. That is, how do we compute m^n? We could multiply by m a total of $n - 1$ times. Can we do better? Yes, there is a simple divide and conquer approach that we can use. We can recognize that, when n is even, $m^n = m^{n/2}m^{n/2}$. If n is odd, then $m^n = m^{\lfloor n/2 \rfloor}m^{\lfloor n/2 \rfloor}m$. This leads to the following recursive algorithm

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

Function **Power** has recurrence relation

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

whose solution is

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

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

How does this cost compare with the problem size? The original problem size is $\log m + \log n$, and the number of multiplications required is $\log n$. This is far better (in fact, exponentially better) than performing $n - 1$ multiplications.

16.3.2 Largest Common Factor

We will next present Euclid's algorithm for finding the largest common factor (LCF) for two integers. The LCF is the largest integer that divides both inputs evenly.

First we make this observation: If k divides n and m, then k divides $n - m$. We know this is true because if k divides n then $n = ak$ for some integer a, and if k divides m then $m = bk$ for some integer b. So, $LCF(n, m) = LCF(n - m, n) = LCF(m, n - m) = LCF(m, n)$.

Now, for any value n there exists k and l such that

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

From the definition of the $\mod$ function, we can derive the fact that

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

Since the LCF is a factor of both n and m, and since $n = km + l$, the LCF must therefore be a factor of both km and l, and also the largest common factor of each of these terms. As a consequence, $LCF(n, m) = LCF(m, l) = LCF(m, n \bmod m)$.

This observation leads to a simple algorithm. We will assume that $n \geq m$. At each iteration we replace n with m and m with $n \bmod m$ until we have driven m to zero.

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

To determine how expensive this algorithm is, we need to know how much progress we are making at each step. Note that after two iterations, we have replaced n with $n \bmod m$. So the key question becomes: How big is $n \bmod m$ relative to n?

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

Thus, function LCF will halve its first parameter in no more than 2 iterations. The total cost is then $\mathrm{O}(\log n)$.

16.3.3 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.

For simplicity, we will 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 $n/2 \times n/2$. Using

this notation, we can think of matrix multiplication using divide and conquer in the following way:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

Of course, each of the multiplications and additions on the right side of this equation are recursive calls on arrays of half size, and additions of arrays of half size, respectively. The recurrence relation for this algorithm is

$$T(n) = 8T(n/2) + 4(n/2)^2 = \Theta(n^3).$$

This closed form solution can easily be obtained by applying the Master Theorem 14.1.

Strassen's algorithm carefully rearranges the way that the various terms are multiplied and added together. It does so 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 different 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:

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

With a little effort, you should be able to verify that this peculiar combination of operations does in fact produce the correct answer!

Now, looking at the list of operations to compute the s factors, and then counting the additions/subtractions needed to put them together to get the final answers, we see that we need a total of seven (array) multiplications and 18 (array) additions/subtractions to do the job. This leads to the recurrence

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

We obtained this closed form solution again by applying the Master Theorem.

Unfortunately, while Strassen's algorithm does in fact reduce the asymptotic complexity over the standard algorithm, the cost of the large number of addition and subtraction operations raises the constant factor involved considerably. This means that an extremely large array size is required to make Strassen's algorithm practical in real applications.

16.3.4 Random Numbers

The success of randomized algorithms such as were presented in Section 16.2 depend on having access to a good random number generator. While modern compilers are likely to include a random number generator that is good enough for most purposes, it is helpful to understand how they work, and to even be able to construct your own in case you don't trust the one provided. This is easy to do.

First, let us consider what a random sequence. From the following list, which appears to be a sequence of "random" numbers?

- 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, ...

In fact, all three happen to be the beginning of a some sequence in which one could continue the pattern to generate more values (in case you do not recognize it, the third one is the initial digits of the irrational constant e). Viewed as a series of digits, ideally every possible sequence has equal probability of being generated (even the three sequences above). In fact, definitions of randomness generally have features such as:

- One cannot predict the next item. The series is **unpredictable**.
- The series cannot be described more briefly than simply listing it out. This is the **equidistribution** property.

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.[1] The most commonly used approach historically is known as the **Linear Congruential Method** (LCM). The LCM method is quite simple. We begin by picking a **seed** that we will call $r(1)$. Then, we can compute successive terms as follows.

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

where b and t are constants.

[1] Another approach is based on using a computer chip that generates random numbers resulting from "thermal noise" in the system. Time will tell if this approach replaces deterministic approaches.

By definition of the mod function, all generated numbers must be in the range 0 to $t-1$. Now, consider what happens when $r(i) = r(j)$ for values i and j. Of course then $r(i+1) = r(j+1)$ which means that we have a repeating cycle.

Since the values coming out of the random number generator are between 0 and $t-1$, the longest cycle that we can hope for has length t. In fact, since $r(0) = 0$, it cannot even be quite this long. It turns out that to get a good result, it is crucial to pick good values for both b and t. To see why, consider the following example.

Example 16.4 Given a t value of 13, we can get very different results depending on the b value that we pick, in ways that are hard to predict.

$r(i) = 6r(i-1) \bmod 13 =$
..., 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, ...

Clearly, a b value of 5 is far inferior to b values of 6 or 7 in this example.

If you would like to write a simple LCM random number generator of your own, an effective one can be made with the following formula.

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

16.3.5 The Fast Fourier Transform

As noted at the beginning of this section, multiplication is considerably more difficult than addition. The cost to multiply two n-bit numbers directly is $\mathrm{O}(n^2)$, while addition of two n-bit numbers is $\mathrm{O}(n)$.

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 because it is a physical part of the slide rule. Thus, the entire multiplication process can be done cheaply via a reduction to addition. In the days before electronic calculators, slide rules were routinely used by scientists and engineers to do basic calculations of this nature.

Now consider the problem of multiplying polynomials. A vector $\mathbf{a}$ of n values can uniquely represent a polynomial of degree $n-1$, expressed as

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

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.

Example 16.5 Polynomial A: $x^2 + 1$.

Polynomial B: $2x^2 - x + 1$.

Polynomial AB: $2x^4 - x^3 + 3x^2 - x + 1$.

When we multiply the evaluations of A and B at points 0, 1, and -1, we get the following results.

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

These results are the same as when we evaluate polynomial AB at these points.

Note that evaluating any polynomial at 0 is easy. If we evaluate at 1 and -1, we can share a lot of the work between the two evaluations. But we would need five points to nail down polynomial AB, since it is a degree-4 polynomial. Fortunately, we can speed processing for any pair of values c and $-c$. This seems to indicate some promising ways to speed up the process of evaluating polynomials. But, evaluating two points in roughly the same time as evaluating one point only speeds the process by a constant factor. Is there some way to generalized these

observations to speed things up further? And even if we do find a way to evaluate many points quickly, we will also need to interpolate the five values to get the coefficients of AB back.

So we see that we could multiply two polynomials in less than $\Theta(n^2)$ operations *if* a fast way could be found to do evaluation/interpolation of $2n-1$ points. Before considering further how this might be done, first observe again the relationship between evaluating a polynomial at values c and $-c$. 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 c and $-c$, we get

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

Thus, we only need to compute the Es and Os 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. In particular, we want to take advantage of symmetries, such as the one we see for evaluating x and $-x$. But we need to find even more symmetries between points if we want to do more than cut the work in half. We have to find symmetries not just between pairs of values, but also further symmetries between pairs of pairs, and then pairs of pairs of pairs, and so on.

Recall that a **complex** number z has a real component and an imaginary component. We can consider the position of z on a number line if we use the y dimension for the imaginary component. Now, we will define 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 called the **nth roots of unity**. For example, when $n = 4$, then $z = i$ or $z = -i$. In general, we have the identities $e^{i\pi} = -1$, and $z^j = e^{2\pi ij/n} = -1^{2j/n}$. The significance is that we can find as many points on a unit circle as we would need (see Figure 16.6). But these points are special in that they will allow us to do just the right computation necessary to get the needed symmetries to speed up the overall process of evaluating many points at once.

The next step is to define how the computation is done. Define an $n \times n$ matrix A_z with row i and column j as

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

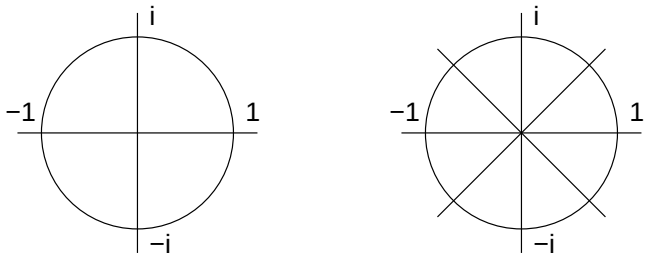

Figure 16.6 Examples of the 4th and 8th roots of unity.

The idea is that there is a row for each root (row i for z^i) while the columns correspond to the power of the exponent of the x value in the polynomial. For example, when $n = 4$ we have $z = i$. Thus, the A_z array appears as follows.

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

Let $a = [a_0, a_1, ..., a_{n-1}]^T$ be a vector that stores the coefficients for the polynomial being evaluated. We can then do the calculations to evaluate the polynomial at the nth roots of unity by multiplying the A_z matrix by the coefficient vector. The resulting vector F_z is called the Discrete Fourier Transform for the polynomial.

$$F_z = A_z a = b.$$

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

When $n = 8$, then $z = \sqrt{i}$, since $\sqrt{i}^8 = 1$. So, the corresponding matrix is as follows.

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

We still have two problems. We need to be able to multiply this matrix and the vector faster than just by performing a standard matrix-vector multiplication,

otherwise the cost is still n^2 multiplies to do the evaluation. Even if we can multiply the matrix and vector cheaply, we still need to be able to reverse the process. That is, after transforming the two input polynomials by evaluating them, and then pair-wise multiplying the evaluated points, we must interpolate those points to get the resulting polynomial back that corresponds to multiplying the original input polynomials.

The interpolation step is nearly identical to the evaluation step.

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

We need to find A_z^{-1}. This turns out to be simple to compute, and is defined as follows.

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

In other words, interpolation (the inverse transformation) requires the same computation as evaluation, except that we 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.

If you examine the example A_z matrix for $n = 8$, you should see that there are symmetries within the matrix. For example, the top half is identical to the bottom half with suitable sign changes on some rows and columns. Likewise for the left and right halves. 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. The algorithm is as follows.

```
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;
}
```

Thus, the full process for multiplying polynomials A and B using the Fourier transform is as follows.

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

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

2. Perform **`Fourier_Transform`** on the representations for A and B
3. Pairwise multiply the results to get $2n-1$ values.
4. Perform the inverse **`Fourier_Transform`** to get the $2n-1$ degree polynomial AB.

16.4 Further Reading

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

16.5 Exercises

16.1 Solve Towers of Hanoi using a dynamic programming algorithm.

16.2 There are six possible 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 11.26.

16.4 The implementation for Floyd's algorithm given in Section 16.1.2 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 prove 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.

value	depth
5	2
20	0
30	0
2	0
25	1
26	3
31	0

16.7 If we had a linked list that would never be modified, we can use a simpler approach than the Skip List to speed access. The concept would remain the same in that we add additional pointers to list nodes for efficient access to the ith element. How can we add a second pointer to each element of a singly linked list to allow access to an arbitrary element in $\mathrm{O}(\log n)$ time?

16.8 What is the expected (average) number of pointers for a Skip List node?

16.9 Write a function to remove a node with given value from a Skip List.

16.10 Write a function to find the ith node on a Skip List.

16.6 Projects

16.1 Complete the implementation of the Skip List-based dictionary begun in Section 16.2.2.

16.2 Implement both a standard $\Theta(n^3)$ matrix multiplication algorithm and Strassen's matrix multiplication algorithm (see Exercise 14.16.3.3). Using empirical testing, try to estimate the constant factors for the runtime equations of the two algorithms. How big must n be before Strassen's algorithm becomes more efficient than the standard algorithm?