16

Patterns of Algorithms

This chapter talks about some fundamental patterns of algorithms. We discuss greed algorithms, dynamic programming, randomized algorithms, numerical algorithms, and the concept of a transform.

Section 16.3.1 presents the Skip List, a probabilistic data structure that can be used to implement the dictionary ADT. The Skip List is comparable in complexity to the BST, yet often outperforms the BST, because the Skip List's efficiency is not tied to the values of the dataset being stored.

16.1 Greedy Algorithms

Observe that the following are all greedy algorithms (that work!): Kruskal's MST, Prim's MST, Dijkstra's shortest paths, Huffman's coding algorithm. Various greedy knapsack algorithms, such as Continuous-knapsack problem (see Johnsonbaugh and Schaefer, Sec 7.6).

Could consider greedy algorithms as approximation algorithms for coping with NP comleteness and optimization.

See Papadimitriou and steighlitz, Computational Optimization: Algorithms and Complexity, Prentice Hall, 1982, for more information on greedy algorithms.

See the treatment by Kleinberg & Tardos.

Consider that a heap has a "greedy" definition: The value of any node A is bigger than its children. The BST's definition is that the value of any node A is greater than all nodes in the left subtree, and less than all nodes in the right subtree. If we try a greedy definition (A is greater than its left child and less than its right child), we can get a tree that meets this definition but is not a BST. See the example in Section 5.2.

16.2 Dynamic Programming

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

```
int Fibr(int n) {
  if (n <= 1) 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 Fibonocci number itself, which our analysis showed to be exponential (approximately $n^1.62$) Why is this so expensive? It is expensive primarily because two recursive calls are made by the function, and they are 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.

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

```
long Fibi(int n) {
  long past, prev, curr;
  past = prev = curr = 1;      // curr holds Fib(i)
  for (int i=2; i<=n; i++) {  // Compute next value
    past = prev; prev = curr; // past holds Fib(i-2)
    curr = past + prev;       // prev holds Fib(i-1)
  }
  return curr;
}
```

This issue of recomputing subproblems comes up frequently. In many cases, arbitrary subproblems (or at least a wide variety of subproblems) might need to be recomputed, so that storing subresults in a fixed number of variables will not work. Thus, there are many times where storing a table of subresults can 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 of 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 the recombined into a solution for the problem being solved. Dynamic programming is appropriate whenever the subproblems to be solved are overlapping in some way. Whenever this happens, dynamic programming can be used if 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 more typically implemented by building the table of subproblems from the bottom up. Thus, `Fibi` is actually closer in spirit to dynamic programming than is `Fibrt` even though it doesn't need the actual table.

16.2.1 Knapsack Problem

Knapsack problem: Given an integer capacity K and n items such that item i has integer size k_i, find a subset of the n items whose sizes exactly sum to K, if possible. Formally: Find $S \subset \{1, 2, ..., n\}$ such that

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

Example: $K = 163$ 10 items of sizes 4, 9, 15, 19, 27, 44, 54, 68, 73, 101. What if K is 164?

Instead of parameterizing problem just by n, parameterize with n and K. $P(n, K)$ is the problem with n items and capacity K.

Think about divide and conquer (alternatively, induction). What if we know how to solve $P(n-1, K)$? If $P(n-1, K)$ has a solution, then it is a solution for $P(n, K)$. Otherwise, $P(n, K)$ has a solution $\Leftrightarrow P(n-1, K-k_n)$ has a solution.

What if we know how to solve $P(n-1,k)$ for $0 \leq k \leq K$? Cost: $T(n) = 2T(n-1)+c$. $T(n) = \Theta(2^n)$.

BUT... there are only $n(K+1)$ subproblems to solve! Clearly, there are many subproblems being solved repeatedly. Store a $n \times K+1$ matrix to contain the solutions for all $P(i,k)$. Fill in the rows from $i=0$ to n, left to right.

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.

Cost: $\Theta(nK)$.

Example 16.1 Knapsack Example: $K=10$. Five items: 9, 2, 7, 4, 1.

	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.

Example: $M(3,9)$ contains O because $P(2,9)$ has a solution. It contains I because $P(2,2) = P(2,9-7)$ has a solution. How can we find a solution to $P(5,10)$? How can we find ALL solutions to $P(5,10)$?

16.2.2 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).

One solution is to run Dijkstra's algorithm $|\mathbf{V}|$ times, each time computing the shortest path from a different start vertex. If **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.

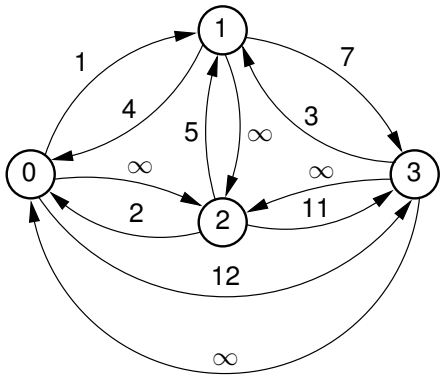

Figure 16.1 An example of k-paths in Floyd's algorithm. Path 1, 3 is a 0-path by definition. Path 3, 0, 2 is not a 0-path, but it is a 1-path (as well as a 2-path, a 3-path, and a 4-path) because the largest intermediate vertex is 0. Path 1, 3, 2 is a 4-path, but not a 3-path because the intermediate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

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. 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++)
      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.

16.3 Randomized Algorithms

What if we settle for the "approximate best?" Types of guarentees, given that the algorithm produces X and the best is Y:

1. $X = Y$.
2. X's rank is "close to" Y's rank:

$$rank(X) \leq rank(Y) + \text{ "small".}$$

3. X is "usually" Y.

$$\mathbf{P}(X = Y) \geq \text{ "large".}$$

4. X's rank is "usually" "close" to Y's rank.

We often give such algorithms names:

1. Exact or deterministic algorithm.
2. Approximation algorithm.
3. Probabilistic algorithm.
4. Heuristic.

We can also sacrifice reliability for speed:

1. We find the best, "usually" fast.
2. We find the best fast, or we don't get an answer at all (but fast).

Choose m elements at random, and pick the best.

- For large n, if $m = \log n$, the answer is pretty good.
- Cost is $m - 1$.

- Rank is $\frac{mn}{m+1}$.

Probabilistic algorithms include steps that are affected by **random** events.

Problem: Pick one number in the upper half of the values in a set. Pick maximum: $n-1$ comparisons. Pick maximum from just over 1/2 of the elements: $n/2$ comparisons. Can we do better? Not if we want a **guarantee**.

Probablilistic algorithm: Pick 2 numbers and choose the greater. This will be in the upper half with probability 3/4. Not good enough? Pick more numbers! For k numbers, greatest is in upper half with probability $1-2^{-k}$.

Monte Carlo Algorithm: Good running time, result not guaranteed. Las Vegas Algorithm: Result guaranteed, but not the running time.

Prime numbers: How do we tell if a number is prime? One approach is the prime sieve: Test all prime up to $\lfloor\sqrt{n}\rfloor$. This requires up to $\lfloor\sqrt{n}\rfloor - 1$ divisions. How does this compare to the input size?

Note that it is easy to check the number of times 2 divides n for the binary representation What about 3? What if n is represented in trinary?

Is there a polynomial time algorithm for finding primes?

Some useful theorems from Number Theory: **Prime Number Theorem**: The number of primes less than n is (approximately)

$$\frac{n}{\ln n}$$

The average distance between primes is $\ln n$. **Prime Factors Distribution Theorem**: For large n, on average, n has about $\ln\ln n$ different prime factors with a standard deviation of $\sqrt{\ln\ln n}$.

To prove that a number is composite, need only one factor. What does it take to prove that a number is prime? Do we need to check all $\sqrt{n}$ candidates?

Some probablistic algorithms:

- Prime(n) = FALSE.
- With probability $1/\ln n$, Prime(n) = TRUE.
- Pick a number m between 2 and $\sqrt{n}$. Say n is prime iff m does not divide n.

Using number theory, we can create a cheap test that will determine that a number is composite (if it is) 50% of the time. Algorithm:

```
Prime(n) {
  for(i=0; i<COMFORT; i++)
    if !CHEAPTEST(n)
      return FALSE;
  return TRUE;
}
```

Of course, this does nothing to help you find the factors!

16.3.1 Skip Lists

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. Imagine that we add a pointer to every other node that lets us skip alternating nodes, as shown in Figure 16.2(b). Define nodes with only a single pointer as level 0 Skip List nodes, while nodes with two pointers are level 1 Skip List nodes.

To search, follow the level 1 pointers until a value greater than the search key has been found, 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 $\Theta(\log n)$.

To implement Skip Lists, we 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

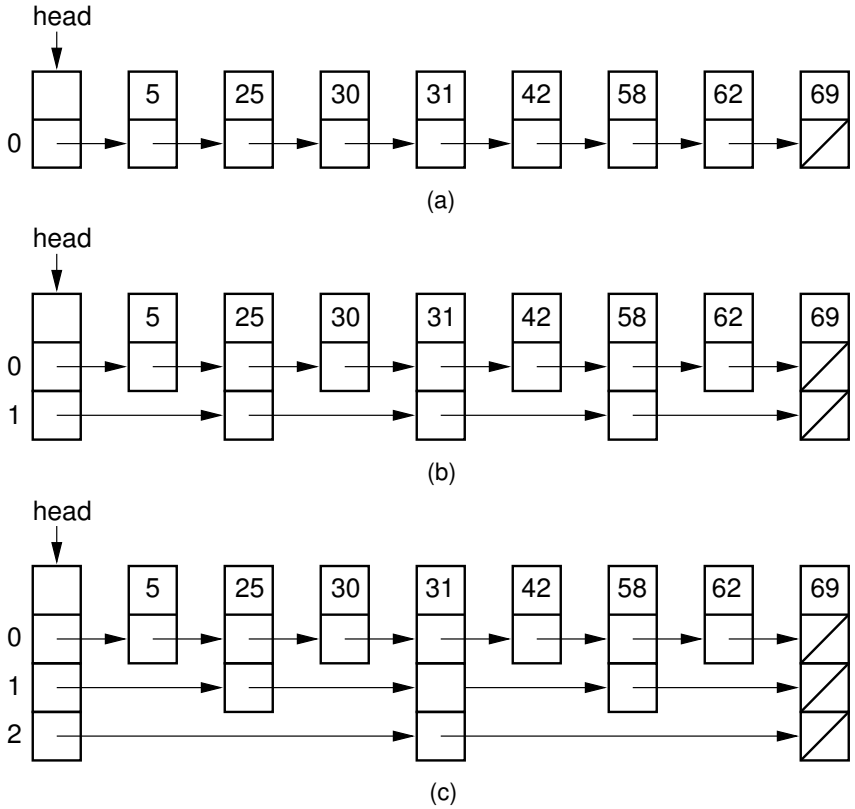

Figure 16.2 Illustration of the Skip List concept. (a) A simple linked list. (b) Augmenting the linked list with additional pointers at every other node. To find the node with key value 62, we visit the nodes with values 25, 31, 58, and 69, then we move from the node with key value 58 to the one with 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.

List class definition includes data member **`level`** that stores the highest level for any node currently in the Skip List. The Skip List is assumed to store 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.

```
public E find(K searchKey) { // Skiplist Search
  SkipNode<K,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.

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.

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

```
/** Insert a record into the skiplist */
public void insert(K 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<K,E>[] update =
         (SkipNode<K,E>[])new SkipNode[level+1];
  SkipNode<K,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<K,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.

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

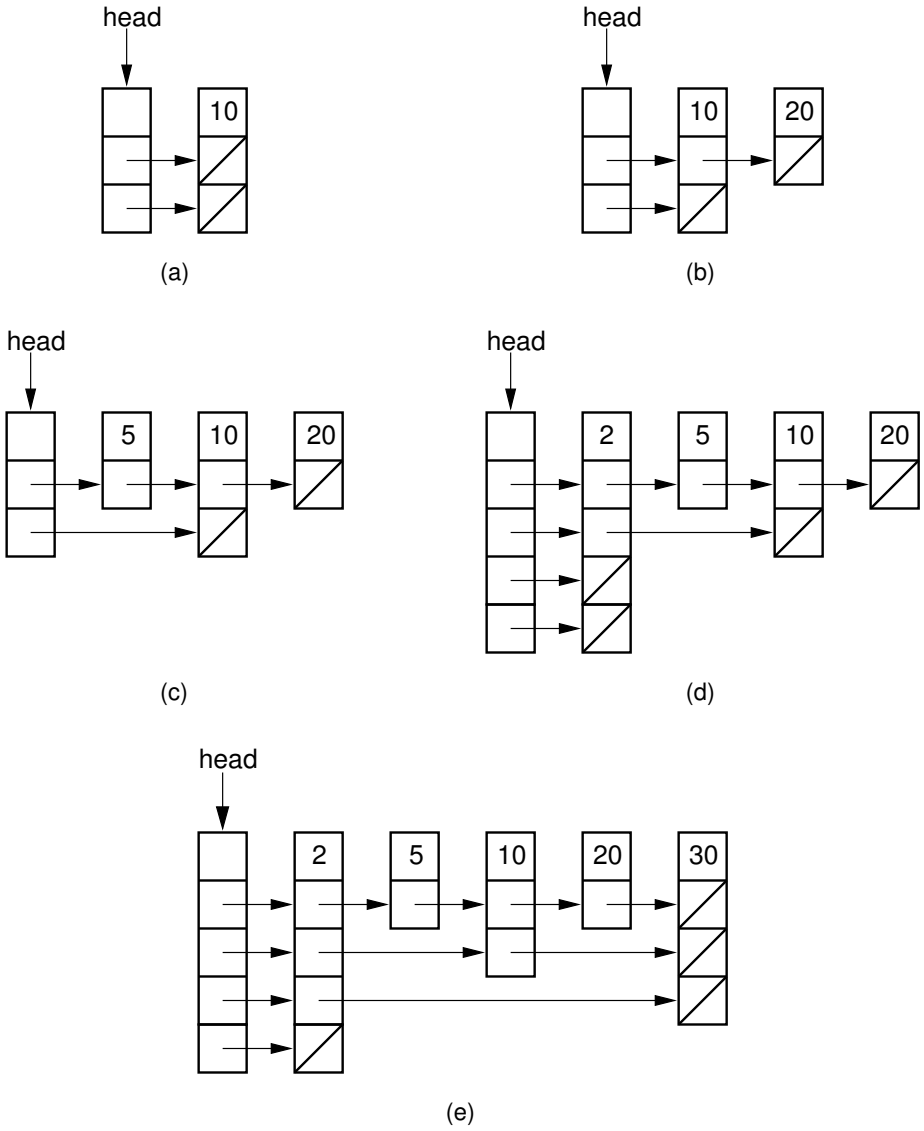

Figure 16.5 Illustration of Skip List insertion. (a) The Skip List after inserting initial value 10 at level 1. (b) The Skip List after inserting value 20 at level 0. (c) The Skip List after inserting value 5 at level 0. (d) The Skip List after inserting value 2 at level 3. (e) The final Skip List after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

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, 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 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:

$$m_1 = (a_{12} - a_{22})(b_{21} + b_{22})$$

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

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, because the cost to multiply two n-bit numbers directly is $\mathrm{O}(n^2)$, 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 because 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 we will 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 could 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.

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.

16.7 Projects

16.1 Complete the implementation of the Skip List-based dictionary begun in Section 16.3.1.

16.2 Implement both a standard $\Theta(n^3)$ matrix multiplication algorithm and Strassen's matrix multiplication algorithm (see Exercise 14.16.4.3). 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?