Lecture 6

Complete Search Review

Let L denote an array of integers, and let f have be a function with prototype **void** f(**int** [] arr).

1. Give recursive code that will process every subset of L and count how many have sum k.
2. Give recursive code that will call the function f one time for each permutation of L (can assume entries are distinct).
3. Give iterative code for calling f once for each permutation of L in lexicographic order.
4. Give recursive code that will call the function f one time for each permutation of L in lexicographic order.
5. Give recursive code that will call f once for each subsequence of L of length 4.
6. Give iterative code that will call f once for each subsequence of L of length 4.

Solutions

1.

```
//Call initially with parameters (L, 0, 0)
static int countSubsets(int[] L, int idx, int sum, int k)
{
  if (idx == L.length) return sum == k ? 1 : 0;
  return countSubsets(L,idx+1,sum,k)
        +countSubsets(L,idx+1,sum+L[idx],k);
}
```

2.

```
static void swap(int[] L, int a, int b)
{
  int tmp = L[a]; L[a] = L[b]; L[b] = tmp;
}
//Call initially with parameters (L, 0)
//Assumes elements of L are distinct (otherwise permutations
   will be repeated)
static void rec(int[] L, int pos)
{
    if (pos == L.length) f(L);
    else
    {
        for (int i = pos; i < L.length; ++i)
        {
            swap(L,pos,i);
            rec(L,pos+1);
            swap(L,pos,i);
        }
    }
}
```

3. The idea is as follows. As we are looking for the next permutation in lexicographic order, we want to change our current permutation (stored in L) at the rightmost possible point. To do this, we look for the longest suffix of L that is in descending order. No pair of elements from such a suffix can be swapped to yield a string that succeeds the current in the lexicographic order. Once we find the first index i that isn't part of a descending suffix, we know which element will be changed. At this point, we swap the element in index i with the smallest value after it that is greater. This operations leaves the suffix after i in descending order. We then reverse the descending suffix to return it to sorted order. The amortized runtime (while iterating over all permtutations) is constant.

```
//Reverse entries in inclusive range [a,b]
static void reverse(int[] L, int a, int b)
{
  while (a < b) swap(L,a++,b--);
}
//Should be initially called with a sorted list
//Changes L to the next permutation in lexicographic order
//Returns false when the list has returned to being sorted
static boolean nextPerm(int[] L)
{
  if (L.length == 1) return false;
  int R = L.length-2;
  while (R>=0)
  {
    if (L[R] >= L[R+1]) --R;
    else
    {
      int i = L.length-1;
      while (L[R] >= L[i]) --i;
      swap(L,R,i);
      reverse(L,R+1,L.length-1);
      return true;
    }
  }
  reverse(L,0,L.length-1);
  return false;
}
//Iterate over the permutations
static void iterate(int[] L)
{
  Arrays.sort(L);
  do
  {
    f(L);
  } while (nextPerm(L));
}
```

4.

```
//Assumes L is sorted; call initially as recLex(L,0)
static void recLex(int[] L, int pos)
{
    if (pos == L.length) f(L);
    else
    {
        recLex(L,pos+1);
        for (int i = pos+1; i < L.length; ++i)
        {
            if (L[i] == L[pos]) continue;
            swap(L,pos,i);
            recLex(L,pos+1);
        }
        int tmp = L[pos];
        for (int i = pos+1; i < L.length; ++i) L[i-1] = L[i];
        L[L.length-1] = tmp;
    }
}
```

5.

```
//Call initially with parameters (L,new int[4],0,0)
static void rec4(int[] L, int[] buf, int pos, int cnt)
{
    if (cnt == 4) f(buf);
    else if (pos < L.length)
    {
        buf[cnt] = L[pos];
        rec4(L,buf,pos+1,cnt+1);
        rec4(L,buf,pos+1,cnt);
    }
}
```

6.

```
static void iter4(int[] L)
{
    int[] buf = new int[4];
    for (buf[0] = 0; buf[0] < L.length; ++buf[0])
      for (buf[1] = buf[0]+1; buf[1] < L.length; ++buf[1])
        for (buf[2] = buf[1]+1; buf[2] < L.length; ++buf[2])
          for (buf[3] = buf[2]+1; buf[3] < L.length; ++buf[3])
            f(buf);
}
```

Greedy Algorithms

To determine whether complete search will solve your problem, you must address the following questions:

1. How can I traverse the search space?
2. How long will the complete search take?
3. How can I implement the complete search?

Usually the most pertinent question is whether the complete search will run in time. If yes, then complete search is usually the way to go. In contrast, the questions for a greedy algorithm are:

1. What is my greedy choice function?
2. How long will the greedy algorithm take?
3. Is it correct?
4. How can I implement my greedy algorithm?

For greedy algorithms, the questions that usually matter most are what is the choice function, and whether it is correct. To better answer this question, we will look at some examples of greedy algorithms.

Exercises

1. You are given a list of tasks, with task i requiring time interval $[a_i, b_i)$, and the requirement that no two tasks can be scheduled at the same time. Give an algorithm to compute the largest number of tasks that can be completed.
2. You are given a list of objects, with object i having weight w_i and value v_i. Assuming you can take fractional amounts of each object, and your sack has capacity C, compute the largest value you can carry.

3. You are given a list of tasks, with task i having duration d_i in days. For each day before task i is started, a cost s_i is paid. Choose an ordering of the tasks that minimizes the total cost paid.

4. Give a greedy algorithm for computing the minimum/maximum spanning tree.

5. Given infinitely many coins with denominations in the set $\{50, 25, 10, 5, 1\}$, give an algorithm for making change for the amount A that uses the least number of coins.

Solutions

1. Sort tasks in ascending order by b_i breaking ties by ascending a_i-values. Always choose the earlier task in the sorted order that is feasible. Now we must show this is correct. Let $P(t)$ denote the maximum number of tasks that can be assigned using times in $[t, \infty)$. Note that $P(s) \geq P(t)$ if $s \leq t$. Suppose we are choosing between $[a_i, b_i)$ and $[a_j, b_j)$ as our earlier task with $b_i \leq b_j$. Then $P(b_i) \geq P(b_j)$ making the first interval at least as good.

2. Sort tasks in descending order by v_i/w_i, and then use objects in order taking as much as possible of each. To show this is optimal, suppose $v_i/w_i \geq v_j/w_j$, you have taken a positive amount of object j, and you have not taken all of object i. Then trading object j for object i cannot disimprove the total value.

3. Sort tasks in descending order by s_j/d_j. To see this is optimal, suppose you have an ordering of the tasks with task i immediately preceding task j. The change in value of swapping these tasks is $s_i d_j - s_j d_i$. This is non-negative precisely when $s_i/d_i \leq s_j/d_j$.

4. We do the minimum case. Sort the edges by weight. Only add an edge if it doesn't create a cycle in the forest of edges already added. Suppose this algorithm is not optimal on some input. Then there must be some first edge e added such that the resulting forest cannot be completed into a minimum spanning tree, but there is a MST T that can be constructed from the edges added before e. Then adding e to T creates a cycle, where at least one edge in the cycle has value greater than or equal to e (as adding e didn't create a cycle in our algorithm). Adding e and removing this larger valued edge creates a tree that is at least as good. Implementation details will be discussed when we cover graph algorithms.

5. Sort the coins in descending order, and use as many as possible of each coin before moving to the next coin. Now we prove this is optimal. Let $50 = c_1 > \cdots > c_5 = 1$ be the coin values. Assume we have created change for the amount A only using coins strictly smaller than c_i, where $A \geq c_i$. If some of the coins used total to c_i, then we can replace them with c_i and obtain a better result. Otherwise we have the following cases:

(a) $c_i = 50$ and a quarter, and 3 dimes are used: replace with a 50 cent piece and a nickel.

(b) $c_i = 25$ and 3 dimes are used: replace with a quarter and a nickel.

Divide and Conquer

Simply put, with divide and conquer algorithms we break a problem into multiple smaller subproblems, solve those subproblems, and then combine the results. We will look at some examples of divide and conquer algorithms, and then focus on one particular kind, binary search.

One classic example is merge sort. In each step we divide our list into two nearly equal parts, recursively merge sort each part, and then merge the two sorted halves to complete the sort. Quick sort also has this flavor. In quick sort we choose a pivot (maybe randomly), partition around the pivot, and then recursively quick sort the two smaller parts.

Here is another way that divide and conquer type thinking (also called meet-in-the-middle) is sometimes used. Suppose we have a list of $n \leq 30$ signed integers and a 64-bit integer L. We want to compute whether some sublist (not necessarily contiguous) sums to L. We have already seen the brute force solution to this problem, but with 30 integers it will be too slow. A better method is to divide the problem into two sublists of size $n/2$, and then compute all possible sums that can occur for each half. After sorting these generated sums, we can loop over them and determine the result. The runtime of this algorithm is $O(2^{n/2} \lg(2^{n/2})) = O(n2^{n/2})$ but can be improved to $O(2^{n/2})$ if radix sort is used.

As a final example, suppose we want to multiply two big integers a, b of length (in bits) $2m$. Then we can write

$$a = a_h \cdot 2^m + a_l, \quad b = b_h \cdot 2^m + b_l.$$

Multiplying gives

$$ab = a_h b_h 2^{2m} + (a_h b_l + a_l b_h)2^m + a_l b_l.$$

Note that

$$a_h b_l + a_l b_h = (a_h + a_l)(b_h + b_l) - a_h b_h - a_l b_l.$$

Thus we can compute ab recursively using 3 multiplications with operands that are half as big. This gives a $O(n^{\lg 3})$ where $\lg 3 \approx 1.58$ which is better than the naive $O(n^2)$ algorithm. This is called the Karatsuba algorithm. For larger numbers, there is a different divide and conquer algorithm called Schonhage-Strassen that uses the Fast Fourier Transform and runs in $O(n \lg n \lg \lg n)$ time.

Binary Search

The standard application of binary search is to look for an element in a sorted array. This can be done in Java with Arrays.binarySearch and Collections.binarySearch and in C++ with lower_bound and upper_bound. One issue with using the Java binary search function is when there are duplicates in your array. Suppose $L = [1, 2, 2, 2, 2, 2, 2, 2, 3]$. If you call the Java

binary search function looking for 2, it will return the index of one of the twos. There are problems when you absolutely need to find the first or last occurrence of a given index. Hence, it is useful to be able to write binary search yourself.

That said, there are many other ways binary search can be applied to solve problems. For example, suppose you wanted to find a root of $f(x) = 2x^4 + x^3 - e^x + \log(x)$ on $[1, \infty)$. Since $f(1) > 0$ and $\lim_{x\to\infty} f(x) = -\infty$ the intermediate value theorem shows there must be a root. As $f(20) < 0$ we can binary search for a root in the interval $[1, 20]$, always choosing the new interval such that the f-images of the endpoints have opposite signs. We continue the binary search until the precision is satisfactory.

Another useful way to apply binary search is to remove one dimension of a complex problem. Consider the following problem: You are given a list of events that occur during your trip through the desert. Each event has the time that it occurs, and one of the following:

1. Spring a leak causing you to lose gas at 1 unit per minute (multiple gas leaks can accumulate).
2. Find a mechanic (fixes all leaks).
3. Find a gas station (fills tank).
4. Reach end of trip.

Compute the minimum gas tank size required for the journey. To solve this problem, define a function $f(t)$ which is 1 if the gas tank size t is sufficient to complete the trip, or 0 otherwise. Then f is a monotonic function of t, and thus we can binary search for the smallest value of t such that $f(t) = 1$. By introducing a binary search we turned a computation problem into a simulation problem.

As our final example, we look at the cool problem stated in our textbook. Suppose you are given a tree (up to 80K nodes) where each node has a value greater than its parent. You are also given a list of queries per tree (up to 20K queries), where each query has a node n and a value P. The goal is to find the ancestor of n closest to the root that has value at least P. The idea here is to perform a traversal of the tree keeping track of the path to the root at all times during the traversal. At each node, we find all queries corresponding to that node, and then binary search along the path we are maintaining.

Exercises

Let L be an array of integers.

1. Compute the maximum value in L using divide and conquer.
2. Implement binary search to find the first occurrence of k in L, and a second implementation to find the last occurrence.
3. Given k, compute the kth (0-based) smallest element of L.

4. Assuming L has nonnegative entries, partition L into k contiguous subintervals so that the maximum sum over any of the subintervals is minimized.

5. Suppose you have access to a function **double** f(**double** x) taking inputs from the interval $[0, 100]$. It is given that f (which may be discontinuous) is increasing on $[0, c]$ and decreasing on $[c, 100]$ for some unknown $c \in [0, 100]$. Find c accurate to 6 digits after the decimal point.

6. You run a widget factory, where a widget is made from parts A, B, and C. Each widget needs A, B, C in quantities n_a, n_b, n_c, respectively. If p_a, p_b, p_c are the prices for purchasing a single A, B, C, you start with s_a, s_b, s_c of each part, and you have d dollars at your disposal, compute the maximum number of widgets that can be built.

Solutions

1. Pseudocode follows:

```
int findMax(int[] arr) { return findMax(arr,0,arr.length-1); }
int findMax(int[] arr, int L, int R)
{
  if (L == R) return arr[L];
  int M = (L+R)/2; //M < R always
  return Math.max(findMax(arr,L,M),findMax(arr,M+1,R));
}
```

   The runtime is $O(n)$ (same as iterating through the elements).

2. Pseudocode for first occurrence:

```
//Find first occurrence of k in arr with indices in range [L,R]
int binSearch(int[] arr, int k, int L, int R)
{
  if (L == R) return arr[k] == k ? L : -1;
  int M = (L+R)/2; //M < R
  return arr[M] <= k ? binSearch(arr,k,L,M) :
     binSearch(arr,k,M+1,R);
}
```

   Pseudocode for last occurrence:

```
//Find first occurrence of k in arr with indices in range [L,R]
int binSearch(int[] arr, int k, int L, int R)
{
  if (L == R) return arr[k] == k ? L : -1;
  int M = (L+R+1)/2; //M > L
  return arr[M] >= k ? binSearch(arr,k,M,R) :
     binSearch(arr,k,L,M-1);
}
```

For completeness, here is an implementation of first occurrence using a while loop:

```
while (L < R)
{
  int M = (L+R)/2;
  if (arr[M] <= k) R = M;
  else L=M+1;
}
//Check arr[L]
```

3. By sorting we can do this in $O(n \lg n)$. With a priority queue we can do it in $O(k \lg n + n)$ (the n is to build the queue). The algorithm below uses divide and conquer with the quick sort partition algorithm to achieve an expected $O(n)$ runtime.

```
Random ran = new Random();
static int ranPartition(int[] arr, int L, int R)
{
    int len = R - L + 1, p = ran.nextInt(len)+L;
    int piv = arr[p];
    swap(arr,R,p);
    int l = L, r = R-1;
    while (true)
    {
        while (l <= R && arr[l] < piv) l++;
        while (L <= r && arr[r] > piv) r--;
        if (l >= r)
        {
            if (l >= R) return R;
            else swap(arr,l,R);
            return l;
        }
        swap(arr,l,r);
        l++; r--;
    }
}
//Permutes arr so that arr[i] <= arr[n] for i in [L,n) and
//arr[n] <= arr[i] for i in (n,R]
static void nthElement(int[] arr, int L, int R, int n)
{
    int p = ranPartition(arr,L,R);
    if (n == p) return;
    else if (n < p) nthElement(arr,L,p-1,n);
    else nthElement(arr,p+1,R,n);
}
```

4. Binary search for the correct answer. For each value tested, loop over the array from left to right packing each interval with as many numbers as possible.

5. Pseudocode follows:

```
static double findMax(double L, double R)
{
    if (R–L < 1e–9) return L;
    double F = L + (R–L)/3, S = R – (R–L)/3;
    double fF = f(F), fS = f(S);
    if (fF < fS) return findMax(F,R);
    else return findMax(L,S);
}
```

6. Binary search for the number of widgets you can make. Once you have fixed the amount you need to make, purchase precisely the materials you need.