CSE 101 - ALGORITHMS - SUMMER 2000

Lecture Notes 4

Wednesday, July 12, 2000.

3.3.3 Bucket Sort

Like counting sort and radix sort, becket sort also assumes that the input verfies some property. More precisely, it assumes that the n elements are uniformly distributed over a certain domain; to simplify the exposition we suppose that they are uniformly distributed over the interval $[0, 1)$. Thus, if we divide the interval $[0, 1)$ in n equal-sized subintervals $[0, 1/n)$, $[1/n, 2/n)$, ..., $[(n-1)/n, 1)$ called *buckets*, then we expect only very few numbers to fall into any of the buckets.

Let $B[0, ..., n-1]$ be an array of n lists which are initially empty. Then the following algorithm runs in linear time on the average:

BUCKET-SORT(A)

for $i \leftarrow 1$ **to** n **do**

  insert $A[i]$ at its position into the list $B[\lfloor n \cdot A[i] \rfloor]$

concatenate all the lists $B[0]$, $B[1]$, ..., $B[n-1]$

Exercise 3.1 Illustrate the execution of BUCKET-SORT($\langle 0.87, 0.21, 0.17, 0.92, 0.81, 0.12, 0.67, 0.52, 0.77, 0.85 \rangle$).

Exercise 3.2 Informally analyze BUCKET-SORT.

3.4 Problems Related to Sorting

This section presents two interesting and useful problems which are or seem to be very related to sorting.

3.4.1 Binary Search

Searching is a common operation of databases. The search problem can be formulated in various ways, depending on the type of desired returned information. We only consider an oversimplified version in this subsection:

SEARCH

INPUT: A sequence $A = \langle a_1, a_2, ..., a_n \rangle$ and an element x.

OUTPUT: If x occurs in A then x, otherwise *not_found.*

The method to do it seems obvious: visit each element in A and compare it with x; if such alement is found then return it, otherwise return *not_found*. This algorithm obviously runs in $O(n)$ time, which seems to be good. Unfortunately, linear algorithms are too slow when bilions of items are involved (imagine Yahoo's databases). For that reason, the databases are in general organized in efficient datastructures, thus making operations like search (and many others) run in logarithminc time. We only consider one of the simplest data structures in this subsection, a sorted array. Therefore, let us consider the following more refined problem:

SORTED-SEARCH

INPUT: A *sorted* sequence $A = \langle a_1, a_2, ..., a_n \rangle$ and an element x.

OUTPUT: If x occurs in A then x, otherwise *not_found.*

We can now use a simple degenerated divide and conquer algorithm to solve this problem, which is called *binary search*:

BINARY-SEARCH(A, i, k, x)

if $i > k$ **then return** *not_found*

$j \leftarrow \lfloor \frac{i+k}{2} \rfloor$

if $x = A[j]$ **then return** x

if $x < A[j]$ **then return** BINARY-SEARCH($A, i, j-1, x$)

return BINARY-SEARCH($A, j+1, k, x$)

Then BINARY-SEARCH$(A, 1, n, x)$ is a solution algorithm for the problem **SORTED-SEARCH**. From now on in the course, we'll write BINARY-SEARCH(A, x) instead of BINARY-SEARCH$(A, 1, n, x)$ whenever possible.

Exercise 3.3 Illustrate the execution of BINARY-SEARCH$(A, 3.14)$ for $A = \langle 0.17, 1, 1.27, 2.2, 2.9, 3.14, 3.9, 4.2, 5\rangle$. Do the same thing for $A = \langle 0.17, 1, 1.27, 2.2, 2.9, 3.13, 3.9, 4.2, 5\rangle$.

Exercise 3.4 Why is the input array of BINARY-SEARCH required to be sorted?

Exercise 3.5 Write the recurence for BINARY-SEARCH and then show that its running time is $O(\log n)$.

3.4.2 Medians and Order Statistics

3.5 Exercises with Solutions

Exercise 3.6 Let $A = \langle 3, 9, 5, 3, 1, 4, 8, 7\rangle$. Illustrate the execution of

1. INSERTION-SORT(A);
2. SELECTION-SORT(A);
3. QUICK-SORT$(A, 1, 8)$; do *not* illustrate the execution of PARTITION;
4. MERGE-SORT$(A, 1, 8)$; do *not* illustrate the execution of MERGE;
5. HEAP-SORT(A); do *not* illustrate the executions of BUILD-HEAP and HEAPIFY;
6. COUNTING-SORT$(A, 9)$.

Proof: We only show how the array A is modified. This is enough to show that you understood the sorting algorithms. There were many different solutions in your quizes; I condidered them all correct if it was clear that you understood the algorithms.

1. INSERTION-SORT

   $\langle 3, \underline{9}, 5, 3, 1, 4, 8, 7\rangle$
   $\langle 3, 9, \underline{5}, 3, 1, 4, 8, 7\rangle$
   $\langle 3, 5, 9, \underline{3}, 1, 4, 8, 7\rangle$
   $\langle 3, 3, 5, 9, \underline{1}, 4, 8, 7\rangle$
   $\langle 1, 3, 3, 5, 9, \underline{4}, 8, 7\rangle$
   $\langle 1, 3, 3, 4, 5, 9, \underline{8}, 7\rangle$
   $\langle 1, 3, 3, 4, 5, 8, 9, \underline{7}\rangle$
   $\langle 1, 3, 3, 4, 5, 7, 8, 9\rangle$

2. SELECTION-SORT

   $\langle 3, 9, 5, 3, 1, 4, 8, 7\rangle$
   $\langle \underline{1}, 9, 5, 3, 3, 4, 8, 7\rangle$
   $\langle 1, \underline{5}, 9, 3, 3, 4, 8, 7\rangle$
   $\langle 1, \underline{3}, 9, 5, 3, 4, 8, 7\rangle$
   $\langle 1, 3, \underline{5}, 9, 3, 4, 8, 7\rangle$
   $\langle 1, 3, \underline{3}, 9, 5, 4, 8, 7\rangle$
   $\langle 1, 3, 3, \underline{5}, 9, 4, 8, 7\rangle$
   $\langle 1, 3, 3, \underline{4}, 9, 5, 8, 7\rangle$
   $\langle 1, 3, 3, 4, \underline{5}, 9, 8, 7\rangle$
   $\langle 1, 3, 3, 4, 5, \underline{8}, 9, 7\rangle$
   $\langle 1, 3, 3, 4, 5, \underline{7}, 9, 8\rangle$
   $\langle 1, 3, 3, 4, 5, 7, \underline{8}, 9\rangle$

3. QUICK-SORT

   The array A is first partitioned as (consider that the pivot is $A[1]$) $\langle 1, 3, \underline{3}, 9, 5, 4, 8, 7\rangle$. Then QUICK-SORT$(A, 1, 3)$ and QUICK-SORT$(A, 4, 8)$ are called. The first is not going to change the array A, so we only illustrate the second. The subarray $\langle \underline{9}, 5, 4, 8, 7\rangle$ is again partitioned (the pivot is 9 now) modifying A to $\langle 1, 3, 3, 7, 5, 4, 8, 9\rangle$ and then QUICK-SORT$(A, 4, 7)$ is called. We keep doing this and obtain $\langle 1, 3, 3, 4, 5, \underline{7}, 8, 9\rangle$.

4. MERGE-SORT

   $\langle 3, 9, 5, 3, 1, 4, 8, 7\rangle$
   $\langle 3, 9, 5, 3\rangle$ $\langle 1, 4, 8, 7\rangle$
   $\langle 3, 9\rangle$ $\langle 5, 3\rangle$ $\langle 1, 4\rangle$ $\langle 8, 7\rangle$
   $\langle 3\rangle$ $\langle 9\rangle$ $\langle 5\rangle$ $\langle 3\rangle$ $\langle 1\rangle$ $\langle 4\rangle$ $\langle 8\rangle$ $\langle 7\rangle$
   $\langle 3, 9\rangle$ $\langle 3, 5\rangle$ $\langle 1, 4\rangle$ $\langle 7, 8\rangle$
   $\langle 3, 3, 5, 9\rangle$ $\langle 1, 4, 7, 8\rangle$
   $\langle 1, 3, 3, 4, 5, 7, 8, 9\rangle$

5. HEAP-SORT

   The procedure BUILD-HEAP yields $A = \langle 9, 7, 8, 3, 1, 4, 5, 3\rangle$. Then the following changes generated by swappings and heapifies end up with the sorted array:

   $\langle 3, 7, 8, 3, 1, 4, 5, \underline{9}\rangle$
   $\langle 8, 7, 5, 3, 1, 4, 3, \underline{9}\rangle$
   $\langle 3, 7, 5, 3, 1, 4, \underline{8}, \underline{9}\rangle$
   $\langle 7, 3, 5, 3, 1, 4, \underline{8}, \underline{9}\rangle$
   $\langle 4, 3, 5, 3, 1, \underline{7}, \underline{8}, \underline{9}\rangle$
   $\langle 5, 3, 4, 3, 1, \underline{7}, \underline{8}, \underline{9}\rangle$
   $\langle 1, 3, 4, 3, \underline{5}, \underline{7}, \underline{8}, \underline{9}\rangle$
   $\langle 4, 3, 1, 3, \underline{5}, \underline{7}, \underline{8}, \underline{9}\rangle$
   $\langle 3, 3, 1, \underline{4}, \underline{5}, \underline{7}, \underline{8}, \underline{9}\rangle$
   $\langle 1, 3, \underline{3}, \underline{4}, \underline{5}, \underline{7}, \underline{8}, \underline{9}\rangle$
   $\langle 1, \underline{3}, \underline{3}, \underline{4}, \underline{5}, \underline{7}, \underline{8}, \underline{9}\rangle$
   $\langle \underline{1}, \underline{3}, \underline{3}, \underline{4}, \underline{5}, \underline{7}, \underline{8}, \underline{9}\rangle$

6. COUNTING-SORT

   The frequency array is $F = \langle 1, 0, 2, 1, 1, 0, 1, 1, 1\rangle$. After the next step, it becomes $F = \langle 1, 1, 3, 4, 5, 5, 6, 7, 8\rangle$. Next, we visit the elements in A from the last one toward the first and output them in B according to F, appropriately decreasing the frequencies. We get $B = \langle 1, 3, 3, 4, 5, 7, 8, 9\rangle$ and $F = \langle 0, 1, 1, 3, 4, 5, 5, 6, 7\rangle$.

Exercise 3.7 Insertion sort can be expressed as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

Proof: Let $T(n)$ be the time needed to sort an array of n elements using this recursive version of insertion sort. In the recursive step of this version, the same algorithm is run on an array of $n-1$ elements, and there is an additional step of at most $n-1$ comparison in order to insert the last element input in the sorted array of $n-1$ elements. Then the recurrence for the running time is

$$T(n) = T(n-1) + O(n).$$

Exercise 3.8 **<u>Exercise 2-6. in Skiena</u>** You can use any sorting algorithms presented so far as subroutines.

1. Let S be an unsorted array of n integers. Give an algorithm which finds the pair $x, y \in S$ that *maximizes* $|x-y|$. Your algorithm must run in $O(n)$ worst-case time.

2. Let S be a sorted array of n integers. Give an algorithm which finds the pair $x, y \in S$ that *maximizes* $|x-y|$. Your algorithm must run in $O(1)$ worst-case time.

3. Let S be an unsorted array of n integers. Give an algorithm which finds the pair $x, y \in S$ that *minimizes* $|x-y|$, for $x \neq y$. Your algorithm must run in $O(n \lg n)$ worst-case time.

4. Let S be an sorted array of n integers. Give an algorithm which finds the pair $x, y \in S$ that *minimizes* $|x - y|$, for $x \neq y$. Your algorithm must run in $O(n)$ worst-case time.

Proof:

1. We scan through the array once keeping track of the smallest and the largest integers found so far. At the end we have the smallest and largest integers which when subtracted will get the maximum absolute difference. The worst-case running time of the algorithm is $T(n) = 2n = O(n)$ because we compare each integer in the array with the smallest integer and largest integer found so far.

2. We can let $y = A[1]$ and $x = A[n]$. Since the array is already sorted y will be the smallest element in the array, and x will be the largest element in the array thus maximizing the difference $|x-y|$. Accessing an array requires constant time so the worst-case running time of this algorithm is $O(1)$.

3. We can sort the array using MergeSort which runs in worst-case time $O(n \lg n)$. Then we can traverse array keeping track of the two consecutive array elements which have the smallest difference and are not equal. A single traversal of the array takes worst-case time $O(n)$. The total worst-case time of the algorithm is $O(n \log n)$.

4. Again we traverse the array keeping track of the two consecutive array elements which have the smallest difference and are not equal. The worst-case running time of this algorithm is $O(n)$.

Exercise 3.9 Given an array of real numbers S, find a pair of numbers x, y in S that *minimizes* $|x + y|$. Give your best algorithm for this problem, argue that it is correct and then analyze it. For partial credit you can write an $O(n^2)$ algorithm.

Proof: A solution in $O(n^2)$ time would be to generate all pairs x, y, to calculate $|x + y|$ and to select those that give the minimum.

The $O(n \log n)$ solution based on sorting that I expected is the following:

MIN-SUM(S)

1 sort S by the *abolute value* of its elements

2 $min \leftarrow \infty$

3 **for** $i \leftarrow 1$ **to** $length(S) - 1$

4   **if** $|S[i] + S[i+1]| < min$ **then** $\{min \leftarrow |S[i] + S[i+1]|;\ x \leftarrow S[i];\ y \leftarrow S[i+1]\}$

5 **return** x, y.

The idea is to sort the numbers by their absolute value. This can be easily done by modifying any sorting algorithm such that to compare $|S[i]|$ with $|S[j]|$ instead of $S[i]$ with $S[j]$; the running time remains the same.

This algorithm is correct because if two numbers x and y minimize the expression $|x+y|$, then one of the following cases can appear:

1. x, y are both positive; then x, y must be the smallest two positive numbers, so they occur at consecutive positions in the sorted array.

2. x, y are both negative; then x, y must be the largest two negative numbers, so they are consecutive in the sorted array.

3. One of x, y is positive and the other is negative; then $|x + y| = ||x| - |y||$; but the expression $||x| - |y||$ is minimized by two consecutive elements in the sorted array.

Therefore, the expression $|x + y|$ is minimized by two elements x, y which occur on consecutive positions in the array sorted by absolute value of numbers.

The running time of this algorithm is given by the running time of sorting in step 1, because the other steps take linear time. Hence, the running time is $O(n \log n)$.

Exercise 3.10 (2.5, Skiena) Given two sets S_1 and S_2 (each of size n), and a number x, describe an $O(n \lg n)$ algorithm for finding whether there exists a pair of elements, one from S_1 and one from S_2, that add up to x.

Proof: **(Solution 1)** A $\Theta(n^2)$ algorithm would entail examining each element y_1 in S_1 and determining if there is an element y_2 in S_2 such that $y_1 + y_2 = x$.

FINDSUM(S_1, S_2, x)
for $y_i \in S_1$ **do**
  for $y_j \in S_2$ **do**
    if $y_i + y_j = x$ **then return** (y_i, y_j)

Proof: **(Solution 2)** A more efficient solution takes advantage of sorting as a subroutine. First, we sort the numbers in S_2. Next, we loop through each element y_i in S_1 and then do a binary search on the sorted S_2 for $x - y_i$.

FINDSUM(S_1, S_2, x)
MERGE-SORT($S_2, 1, n$)
for $y_i \in S_1$ do
  $y_j \rightarrow$ BINARY-SEARCH($S_2, x - y_i$)
  if $y_j \neq$ *not_found* **then return** (y_i, y_j)

The MERGE-SORT takes time $\Theta(n \lg n)$. For each element in S_1 the BINARY-SEARCH on S_2 will take $O(\lg n)$. Since this binary search is done n times the for loop requires $O(n \lg n)$ worst-case time. The worst-case time for the entire algorithm is also $O(n \lg n)$.

Exercise 3.11 Give an efficient algorithm for sorting a list of n keys that may each be either 0 or 1. What is the order of the worst-case running time of your algorithm? Write the complete algorithm in pseudo-code. Make sure your algorithm has the stable sorting property.

Proof: Use radix sort (with count sort) for one digit numbers (so $k = 1$). Then the running time is $O(n \cdot k)$, that is, $O(n)$. Alternatively, you can create two arrays, one for zeros and another one for ones, scan the input updating the two arrays, and then append them. The pseudo-code is easy.

Exercise 3.12 (From Neapolitan and Naimipour, p.84. Problem 13.) Write an algorithm that sorts a list of n items by dividing it into three sublists of almost $n/3$ items, sorting each sublist recursively and merging the three sorted sublists. Analyze your algorithm, and give the results using order notation.

Proof: We're doing a three-way merging version of Mergesort. You may write the pseudocode according to your preferred style, as long as it is clear and correct! I have marked comments with a "#" sign.
  Inputs: positive integer n , array of keys S indexed from 1 to n.
  Outputs: the array S containing the keys in nondecreasing order.

procedure MERGESORT3 (n ; **var** S);
  # **var** will ensure that the changes we make to S will be retained.
const
  $third$ = **floor** $(n/3)$;
var
  U: **array** $[1..third]$;
  V: **array** $[1..third]$;
  W: **array** $[1..(n - 2 * third)]$;
  UV: **array** $[1..(2 * third)]$;
    # This array will consist of the keys in both U and V in sorted
    # (nondecreasing) order. This is a stepping stone to merging all three
    # sorted sublists.
begin

if $n = 2$ **then**
sort(S)
# We assume you know how to write the code for this case
# (when S has only two elements).
elseif $n > 1$ **then**
copy $S[1]$ through $S[third]$ to U;
copy $S[third + 1]$ through $S[2 * third]$ to V;
copy $S[2 * third + 1]$ through $S[n]$ to W;
MERGESORT3($third, U$);
MERGESORT3($third, V$);
MERGESORT3($n - 2 * third, W$);
MERGE($third, third, U, V, UV$);
# The inputs to MERGE are: h and m (the lengths of the two sorted # arrays to be merged), the arrays themselves, and the **var**-ed array that # will contain the keys in the two smaller arrays in sorted (non-decreasing) # order. MERGE is $O(h + m - 1)$, which means that it is $O(n)$ in the # larger context of MERGESORT3.
MERGE($2 * third, n - 2 * third, UV, W, S$);
end
end;

The master theorem will aid us in our time complexity analysis. Let us consider W(n), the worst-case time complexity function. The recurrence relation is: $W(n) = 3W(n/3) + O(n)$ (don't worry about floors and ceilings here). Remember that MERGE is $O(n)$, with its worst-case W(n) ϵ $\Theta(n)$; the copy operations are also $O(n)$, if you would like to count those. So, we will use the master theorem with $a = 3, b = 3,$ and $f(n) \epsilon\ \Theta(n)$ and $O(n)$. Then, $n^{log_b a} = n^{log_3 3} = n^1 = n$. We are in case 2 of the master theorem, and thus $W(n) \epsilon\ \Theta(n \lg n)$.

Exercise 3.13 **k-Way Merge Sort**

1. Give an $O(n \log k)$ algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. Analyze the time complexity of your algorithm.

2. Analyze a k-way merge sort algorithm which first splits the input array in k arrays (instead of 2) of size n/k and then merges them. Is it better than merge sort?

Proof:

1. **(Solution 1)** An $O(n \log k)$ algorithm for the problem is to pair the lists and merge each pair, and recursively merge the $k/2$ resulting lists. The time is given by the recurrence $T(n, k) = O(n) + T(n, k/2)$, since the time to perform the merge for each pair of lists is proportional to the sum of the two sizes, so a constant amount of work is done per element. Then unwinding the recurrence gives $\log k$ levels each with $O(n)$ comparisons, for a total time of $O(n \log k)$.

   Proof: **(Solution 2)** Informally, another solution is the following. Given k sorted lists $L_1, \ldots, L_k$ as input, we want our algorithm to output a sorted array A containing all elements in the lists. As in the algorithm for merging two lists, our algorithm will put at each step one element in the final array A, starting with the smallest and ending with the biggest. Then the basic idea is to construct a heap having k elements, one from each list and use it as a priority queue. More precisely, the i-th element in the heap will be the smallest element in the list L_i which has still not been inserted in the final array A. Then at each step our algorithm will do the following: it will extract the minimum from the heap and insert it in array A; then, if the minimum was an element of list L_j, it will insert in the heap the next smallest element in L_j. Notice that to compute whether the minimum of the heap was in the i-th list, we start our algorithm by rewriting each element $L_i[j]$ of the i-th list as a pair $(L_i[j], i)$; that is, by labeling each element of the list with the number of the list to which it belongs.

   This labeling phase takes n steps, since there are n elements to be labelled. Moreover, there are n basic insertion steps since there are n elements to be inserted in A, and for each of these n steps, there is an operation of extraction of the minimum from the heap and an operation of insertion of an element in the heap, which both take $O(\log k)$ time. The time to construct the heap in the first step is $O(k \log k)$. Then, since $k \leq n$, the overall computation time is $O(n \log k)$. The algorithm is the following:

K-MERGE($L_1, \ldots, L_k$)

for $i = 1, \ldots, k$,
    rewrite all elements $L_i[j]$ of the i-th list as a pair $(L_i[j], i)$;
for $i = 1, \ldots, k$,
    Heap-Insert($H, L_i[1]$);   (insert the first element of list L_i in heap H).
    set $ind_i = 2$;
for $i = 1, \ldots, n$,
    $A[i] \leftarrow$ Heap-extract-min(H);
    $if A[i] = (L_h[j], h)$ for some h then
        Heap-Insert($H, L_h[ind_h]$);   (insert the next element of list L_h in heap H).
        set $ind_h \leftarrow ind_h + 1$;
return(A).

2. There will be k subproblems of size n/k and a k-way merging which takes $O(n \cdot \log k)$. Therefore, the recurrence is $T(n) = k \cdot T(n/k) + O(n \cdot \log k)$. Using the tree method, for example, we get $T(n) = h \cdot (n \cdot \log k)$, where h is the height of the tree, that is, $\log_k n$. Hence, $T(n) = n \cdot \log_k n \cdot \log k = n \cdot (\log n / \log k) \cdot \log k = n \cdot \log n$. The conlcusion is that k-way merge sort is not better than the usual merge sort.

Exercise 3.14 (Convex Hull; page 35, Skiena) Given n points in two dimensions, find the convex polygon of smallest area that contains them all.

Proof: Let $(x_1, y_1), .., (x_n, y_n)$ be n points in two dimensions. Very often in geometrical problems, it is a good idea to sort the points by one or both coordinates. In our problem, we sort the points (x_i, y_i) such that either $x_i < x_{i+1}$ or $x_i = x_{i+1}$ and $y_i \leq y_{i+1}$. This sorting assures us that the points are visited from the left to the right and from the bottom to the top when the counter increases from 1 to n.

The strategy to solve this problem is to iteratively obtain the convex hull given by the points $1..i$, where i increases from 2 to n. Notice that the ith point is always a vertex in the convex hull given by the points $1..i$. We have to understand how the convex hull is modified when a new point is added. First, let us consider two arrays $high[1..n]$ and $low[1..n]$ where $high[i]$ and $low[i]$ are going to be the high and the low neighbors of i in the convex hull given by $1..i$, respectively. The key observation is that if there is a point $k \in 1..i-1$ such that k is below the line determined by the points i and $high[k]$ then k cannot be a vertex in the convex hull of $1..i$; similarly, k cannot be a vertex in the convex hull of $1..i$ if k is above the line determined by i and $low[k]$. Summarizing all these up, we get the following algorithm:

Step 1:
Sort the pairs (x_i, y_i) using a standard comparison algorithm (for example MERGE-SORT)
where the order relation is given by: $(x_i, y_i) \leq (x_j, y_j)$ iff $x_i < y_i$ or $x_i = x_j$ and $y_i \leq y_j$,
Step 2:
$high[1..n], low[1..n]$; $high[1] = 0, low[1] = 0$
for $i = 2$ **to** n
    $k = i - 1$
    while BELOW$(k, i, high[k]) = true$
        $k = high[k]$
    $high[i] = k$
    $k = i - 1$
    while ABOVE$(k, i, low[k]) = true$
        $k = low[k]$
    $low[i] = k$

where BELOW(k, i, j) is true iff k is below the line determined by the points i and j, and ABOVE(k, i, j) is true iff k is above the line determined by the points i and j.

So we started with $k = i - 1$ and moved along the high neighbors until the first point k which is a vertex in the convex hull determined by $1..i$ was found; this k is the high neighbor of i. Similarly (but dually) for the low neighbor of i.

Once we have the high and low neighbors of n, we recurrently can obtain and print the convex hull by first following the high path of n and then following the low path of n.

Now, let us analyze the complexity of this algorithm. Step 1 obviously takes $O(n \log n)$. The analyze of Step 2 is more complicated because we have to analyze it globally, counting how many while iterations will be in the whole execution. Notice that once a point k is processed in the first while loop (i.e., BELOW$(k, i, high[k])$ is true), it will not be processed again for a larger i. This is because $high[k]$ will become the high neighbor of i, so k will be hidden for all larger i. Consequently, k will never be visited again within the first while loop. Similarly for the second while loop. Therefore, the while loops are executed $O(n)$ times.

Exercise 3.15 (Onion; 2-17, Skiena) The *onion* of a set of n points is the series of convex polygons that result from finding the convex hull, striping it from the point set, and repeating until no more points are left. Give an $O(n^2)$ algorithm for determining the onion of a point set.

Proof: Make sure that you understand the convex hull problem before you go further.

Following exactly the algorithm described in the text of the problem, it seems that $O(n)$ iterations of convex hull are needed. Since the complexity of convex hull is $O(n \log n)$, the complexity of this problem is going to be $O(n^2 \log n)$. Therefore we have to find something else.

The key point we have to observe is that the points do not have to be sorted again when a new convex hull is generated, because they are already sorted from previous convex hulls. So the points can be sorted only once, at the beginning, and the natural algorithm follows:

Step 1:
Sort the points as in the convex hull problem,

Step 2:
while there still exist points in the set
    (a) generate the current convex hull
    (b) output the convex hull
    (c) strip the convex hull from the point set

The algorithm terminates because each iteration strips some points from the point set. In order for (a) to work properly, we need to make sure that the remained points are still sorted under the ordering described in the convex hull problem after the previous convex hull is removed. Perhaps the easiest way to assure that is to have an array of n flags (boolean values) associated with the points in the set, and in (c) only to change the values of those flags corresponding to the convex hull. Then all n points are scanned in (a) each time when a convex hull is generated, but only those with the proper flag are taken into consideration.

As in the convex hull problem, Step 1 has the complexity $O(n \log n)$. Because the generation of a new convex hull can be done in linear time once the points are sorted (see the convex hull problem), (a) takes $O(n)$; obviously, (b) and (c) need each $O(n)$. So the body of the while loop has a linear complexity. Since each iteration there are some points striped from the point set, the loop has less than n iterations. Therefore, Step 2 has the complexity $O(n^2)$ which is the complexity of the whole algorithm.

Exercise 3.16 (Weighted Median) For n distinct elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the weighted median is the element x_k satisfying

$$\sum_{x_i < x_k} w_i \leq 1/2 \text{ and } \sum_{x_i > x_k} w_i \leq 1/2$$

1. Argue that the median of $x_1, x_2, \cdots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, 2, \cdots, n$.

2. Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.
3. Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm.

Proof:

1.
2.
3. We are given an unsorted list $x_1, ...x_n$ and positive weights $w_1, ..w_n$ with $\sum w_i = 1$, and wish to find an x_i so that $\sum_{j|x_j<x_i} w_j \leq 1/2$ and $\sum_{k|x_k>x_i} w_k \leq 1/2$.

   For a weighted unsorted list $(x_1, w_1), ..(x_n, w_n)$, let $W = \sum_{1 \leq i \leq n} w_i$. We will give an algorithm which solves a more general problem: given a weighted array, and a number $0 \leq \alpha \leq W$ find an x_i so that $\sum_{j|x_j<x_i} w_j \leq \alpha$. and $\sum_{k|x_k>x_i} w_k \leq W - \alpha$.

   We know there is an algorithm $Select$ which can find the j'th largest element of a list of length n in $O(n)$ time. To find the weighted median (or in general, the α-sample), we first find the non-weighted median x_l using Select. Then we divide the list into the subset S of elements smaller than x_l and the subset B of elements bigger than or equal to x_l. Since x_l is the unweighted median, both S and B have at most $n/2$ elements. (If our list contains only a single element, the recursion bottoms out, and we return that element, which is a correct solution since both sums in the two inequalities are over empty sets, hence are 0, and $0 \leq \alpha \leq W$.)

   We then calculate $W_S = \sum_{x_j \in S} w_j$. If $W_S > \alpha$, we recursively call our algorithm on S, α. We output the result of this recursive call, x_k, since x_k satisfies: $\alpha \geq \sum_{j|x_j \in S, x_j < x_k} w_j = sum_{j|x_j<x_k} w_j$, since only elements of S can be smaller than an element of S, and $\sum_{j|x_j>x_k} w_j = \sum_{j|x_j \notin S} w_j + \sum_{j|x_j \in S, x_j > x_k} w_j \leq (W - W_S) + (W_S - \alpha) = W - \alpha$ since all elements not in S are larger than x_k. Otherwise, recursively call the algorithm on $B, \alpha - W_S$. The proof of correctness in this case is similar to the last. Thus, the above algorithm always finds a valid solution.

   Since the time used by the sub-routine select is $O(n)$ as is that to divide the list into B and S and to compute W_S, there is a constant c so that the time taken by the algorithm on an n element input, $T(n)$, is at most cn + the time taken by a recursive call on an input of 1/2 the size. I.e., $T(n) \leq cn + T(n/2)$. Thus $T(n) \leq cn + cn/2 + cn/4 + ... \leq 2cn$, so the algorithm takes $O(n)$ time.

Chapter 4

Dynamic Programming

Dynamic programming is a method related to "divide and conquer", in the sense that in order to calculate a solution of a problem, one first needs to divide it in subproblems of smaller size, solve those subproblems, and then combine their solutions into a solution for the the original problem. However, unlike the divide and conquer algorithms seen so far (merge sort and quick sort) which partitioned the original problems in independent subproblems, dynamic programming is applicable especially when the recursion generates the same subproblems many times, thus appearing that much of the running time is wasted redoing the same computations.

The simple solution proposed by dynamic programming is to *reuse* computation, in the sense that once a subproblem is solved, its result is stored in a table and reused when that subproblem is generated again. An interesting analogy is the cache memory of modern computer systems, where data is available for much faster subsequent accesses after its first access[1].

The discussion above presents dynamic programming in a top-down fashion. However, dynamic programming is better known for its bottom-up approach: it first calculates the solutions of all the subproblems, thus anticipating further requests, and then it uses them to calculate solutions for larger problems (the "divide" stage is hidden). The bottom-up approach uses the same amount of storage but it is slightly more efficient since the function calls are not needed anymore. Additionally, the bottom-up dynamic programming algorithms are more compact and easier to analyze.

Dynamic programming is typically (and hystorically) applied to *optimization problems*, in which an optimal solution is desired. An optimal solution can mean anything, depending on the context; it usually represents a minimum or a maximum value of a certain more or less complex function.

The best way to understand dynamic programming is by examples. For the sake of organization, we'll try to follow the next three steps:

Step 1) Find an appropriate recursive function, like in the "divide and conquer" approaches;

Step 2) Add *momoization* to store the decissions of the subproblems;

Step 3) Find an iterative, bottom-up version of the above.

Any of the steps 2) and 3) is considered good for this class. However, in real practical applications it is almost always better to end up implementing the bottom-up version, like in step 3). Our experience so far is that people are going to feel more confortable to jump directly to step 3) as they understand dynamic programming better.

4.1 Dynamic Programming "Classical" Problems

4.1.1 Fibonacci Numbers

Step 1) We can immediately write the following recurent procedure which is obviously correct:

[1] The computational time is replaced by the time needed to bring the data in the cache in this analogy.

FIBO(n)
if $n < 1$ **then return** 1
return FIBO($n-1$) + FIBO($n-2$)

It does not use auxiliary memory (except the recursion stack), but unfortunately it is too slow.

Exercise 4.1 Show that FIBO runs in time exponential of n.

Step 2) The problem with the recursive function above is that it generates and calculates FIBO(i) many times for each $j > i$, even if it would suffice to calculate it only once. To do this, we need an array $f[0,...,n]$ where to store the values of the Fibonacci terms. Suppose that f initially contains only special symbols ∞.

FIBO(n)
if $f[n] \neq \infty$ **then return** $f[n]$
if $n < 1$ **then return** $f[n] \leftarrow 1$
return $f[n] \leftarrow$ FIBO($n-1$) + FIBO($n-2$)

Exercise 4.2 Show that FIBO of Step 2) runs in time linear of n.

Step 3) We can now completely get rid the recursive function calls as follows:

FIBO(n)
$f[0] \leftarrow f[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
    $f[i] \leftarrow f[i-1] + f[i-2]$
return $f[n]$

4.1.2 Longest Increasing Subsequence

This problem can be formulated as follows:

LONGEST-INCREASING-SUBSEQUENCE

INPUT: An array of real numbers $A = \langle a_1, a_2, ..., a_n \rangle$.

OUTPUT: A longest increasing subsequence of A.

The elements in a subsequence of A need not be neighbors to each other, but they have to appear in the same order in which they appear in A. For example, if $A = \langle 4,2,3,1,9,7,5,8,6 \rangle$ then $\langle 2,3,7,8 \rangle$ is a longest increasing subsequence of A; $\langle 2,3,5,6 \rangle$ is another one.

Step 1) Let us first find a brute-force recursive algorithm based on an optimum condition. Let LIS(A, i) be a function which takes as argument and index i in A and returns a longest increasing subsequence of A that starts with the element $A[i]$. It can be defined as follows:

LIS(A, i)
if $i = n$ **then return** $\langle a_n \rangle$
$local\_longest \leftarrow \langle\rangle$ (the empty sequence)
for $j \leftarrow i+1$ **to** n **do**
    if $A[i] < A[j]$ and $\#local_longest < \#$LIS(A, j) **then** $local\_longest \leftarrow$ LIS(A, j)
return $\langle a_i, local\_longest \rangle$

where $\#sequence$ returns the length of $sequence$.

Then an algorithm that finds the longest increasing subsequence of A would be:

LONGEST-INCREASING-SUBSEQUENCE(A)
$global\_longest \rightarrow \langle\rangle$
for $i \leftarrow 1$ bf to n **to**
  if $\#global_longest < \#$LIS(A, i) **then** $global\_longest \leftarrow$ LIS(A, i)
return $global\_longest$

This algorithm is obviously correct once LIS is correct, and LIS is correct because for each i, a longest subsequence that starts with $A[i]$ verifies the *optimum condition*:

$$\text{LIS}(A, i) = \langle a_i, \max_{i<j, A[i]<A[j]} \text{LIS}(A, j)\rangle.$$

This recursive algorithm can be improved even without momoization, by letting LIS(A, i) return the *length* of the longest increasing subsequence that starts with $A[i]$, instead of the whole subsequence. However, in order to find the subsequence we need to maintain pointers to next elements in longest subsequences. More precisely, let $next[1, ..., n]$ be an array of natural numbers (acting as pointers in A) such that $next[i] = 0$ at initialization. We can now redefine LIS as follows:

LIS(A, i)
if $i = n$ **then return** 1
$local\_max \leftarrow 0$
for $j \leftarrow i + 1$ **to** n **do**
  if $A[i] < A[j]$ and $local\_max <$ LIS(A, j) **then** $\{local\_max \leftarrow$ LIS$(A, j);\ next[i] \leftarrow j\}$
return $1 + local\_max$

LONGEST-INCREASING-SUBSEQUENCE(A) also needs to be redefined to deal with the vector $next$:

LONGEST-INCREASING-SUBSEQUENCE(A)
$global\_max \leftarrow 0;\ index \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  if $global\_max <$ LIS(A, i) **then** $\{global\_max \leftarrow$ LIS$(A, i);\ index \leftarrow i\}$
OUTPUT-SEQUENCE$(A, index)$

where OUTPUT-SEQUENCE$(A, index)$ outputs the computed longest subsequence starting with $A[index]$:

OUTPUT-SEQUENCE$(A, index)$
output $A[index]$
if $next[index] \neq 0$ **then** OUTPUT-SEQUENCE$(A, next[index])$

Exercise 4.3 Show that LONGEST-INCREASING-SUBSEQUENCE(A) runs in exponential time.

Step 2) The procedure LIS(A, i) above is very inefficient because it calculates LIS(A, j) for each $i < j$, each time getting the same result. To calculate it only once, we need to store its value the first time it is calculated and then return the stored value at each further call. Let $l[1, ..., n]$ be an array of n integers, initially ∞. Then LIS can be modified as follows:

LIS(A, i)
if $l[i] \neq \infty$ **then return** $l[i]$
if $i = n$ **then return** $l[n] \leftarrow 1$
$local\_max \leftarrow 0$
for $j \leftarrow i + 1$ **to** n **do**
  if $A[i] < A[j]$ and $local\_max <$ LIS(A, j) **then** $\{local\_max \leftarrow$ LIS$(A, j);\ next[i] \leftarrow j\}$
return $l[i] \leftarrow 1 + local\_max$

Notice that the natural procedure LIS defined in Step 1) suffered very small changes, with a huge impact on the running time.

Exercise 4.4 Show that LONGEST-INCREASING-SUBSEQUENCE(A) runs in polynomial time now.

Step 3) We can get rid of function calls and give a compact efficient iterative solution.

Analyzing the procedure LIS(A, i) defined in Step 2), we can see that despite its top-down fashion, the values stored in $l[1, ..., n]$ are calculated bottom-up, from the last one to the first. On the other hand, the optimum condition presented in Step 1) translates to

$$l[i] = 1 + \max_{i<j, A[i]<A[j]} l[j].$$

Therefore, in order to calculate $l[i]$ one needs all $l[j]$ for $j > i$. All these suggest the following algorithm:

LONGEST-INCREASING-SUBSEQUENCE(A)
$global\_max \leftarrow 0$; $index \leftarrow 0$
for $i \leftarrow n$ **downto** 1 **do**
    $local\_max \leftarrow 0$
    for $j \leftarrow i + 1$ **to** n **do**
        if $A[i] < A[j]$ and $local\_max < l[j]$ **then** $\{local\_max \leftarrow l[j];\ next[i] \leftarrow j\}$
    $l[i] \leftarrow 1 + local\_max$
    if $global\_max < l[i]$ **then** $\{global\_max \leftarrow l[i];\ index \leftarrow i\}$
OUTPUT-SEQUENCE$(A, index)$

This algorithm is correct because it directly implements the optimum condition. Its running time is clearly $O(n^2)$.

Many programmers prefer the iterative bottom-up version and they usually skip the first two steps. However, the first two steps seem more intuitive for beginners who understand recursion well; moreover, the algorithm in Step 2) is usually almost as efficient as its iterative version in practice, staying very close in spirit to its natural recursive ancestor from Step 1).