SIENAcollege
Computer Science

Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2018

Topic Notes: Decrease and Conquer

Our next class of algorithms are the *decrease and conquer* group.

The idea here is that we solve a problem by:

1. Reducing the problem to a smaller instance
2. Solving the smaller instance
3. Modifying the smaller instance solution to be a solution to the original

The main variations on this are:

- Decrease problem size by a constant (often by 1)
- Decrease problem size by a constant factor (often by half)
- Variable size decrease

For example, consider variations on how to compute the value a^n.

The brute-force approach would involve applying the definition and multiplying a by itself, n times.

A *decrease-by-one* approach would reduce the problem to computing the result for the problem of size $n-1$ (computing a^{n-1}) then modifying that to be a solution to the original (by multiplying that result by a).

A *decrease-by-constant-factor* (2, in this case) approach would involve computing $a^{\frac{n}{2}}$ and multiplying that result by itself to compute the answer. The only complication here is that we have to treat odd exponents specially, leading to a rule:

$$a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{if } n \text{ is even} \\ \left(a^{\frac{n-1}{2}}\right)^2 \cdot a & \text{if } n > 1 \text{ and odd} \\ a & \text{if } n = 1 \end{cases}$$

This approach will lead to $O(\log n)$ multiplications.

Insertion Sort

Our decrease-by-one approach to sorting is the *insertion sort*.

The insertion sort sorts an array of n elements by first sorting the first $n-1$ elements, then inserting the last element into its correct position in the array.

ALGORITHM INSERTIONSORT(A)
//Input: an array $A[0..n-1]$
for $i \leftarrow 0..n-1$ **do**
$v \leftarrow A[i]$
$j \leftarrow i-1$
while $j \geq 0$ **and** $A[j] < v$ **do**
$A[j+1] \leftarrow A[j]$
$j \leftarrow j-1$
$A[j+1] \leftarrow v$

This is an in-place sort and is stable.

Our basic operation for this algorithm is the comparison of keys in the while loop.

We do have differences in worst, average, and best case behavior. In the worst case, the while loop always executes as many times as possible. This occurs when each element needs to go all the way at the start of the sorted portion of the array – exactly when the starting array is in reverse sorted order.

The worst case number of comparisons:

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

In the best case, the inner loop needs to do just one comparison, determining that the element is already in its correct position. This happens when the algorithm is presented with already-sorted input. Here, the number of comparisons:

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$$

This behavior is unusual – after all, how often do we attempt to sort an already-sorted array? However, we come close in some very important cases. If we have nearly-sorted data, we have nearly this same performance.

A careful analysis of the average case would result in:

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

Of the simple sorting algorithms (bubble, selection, insertion), insertion sort is considered the best option in general.

Recursive Insertion Sort

As we move into our decrease and conquer and moreso with divide and conquer, we will often find it convenient to consider algorithms in recursive formulations. Let's do that with insertion sort.

Think about what the iterative solution did: it built up larger and larger sorted subarrays by inserting another value into its correct position. After $n-1$ such steps, the entire array is sorted.

To think about this problem recursively, we will, in some sense, think about the problem backwards. Assuming we can obtain a sorted array of size $k-1$ (recursive step), we simply need to insert one element into its correct position to get a sorted array of size k.

ALGORITHM INSERTIONSORT(A)
//Input: an array $A[0..n-1]$
$recInsertionSort(A, n-1)$

ALGORITHM RECINSERTIONSORT(A, max)
//Input: an array $A[0..n-1]$
//Input: upper index limit to sort max
// Base case: a 1-element array
if $max = 0$ **then**
    return
// Recursive case: sort first $max-1$
$RecInsertionSort(A, max-1)$
// now insert max'th in correct location
$v \leftarrow A[max]$
$j \leftarrow max-1$
while $j \geq 0$ **and** $A[j] > v$ **do**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j-1$
$A[j+1] \leftarrow v$

This algorithm performs the same steps as the iterative version, but how could we analyze this? The recursive step doesn't lend itself well to setting up summations. Instead, we will set up and solve a recurrence.

Before solving this one, we will look at a few recurrences in the earlier topic notes on Analysis Fundamentals.

(see those notes)

To analyze the recursive insertion sort, we will count comparisons between array elements, and in the worst case – when the `while` loop runs all the way down to when `j` is -1.

The base case is when we are sorting a 1-element array, and that takes 0 comparisons, so the stopping condition for our recurrence is $C(1) = 0$.

In the recursive case, the number of comparisons for insertion sort on an array of size n is the

comparisons in the recursive call, which solves the problem for an array of size $n-1$, plus the number of comparisons done to complete the sort by inserting the nth element into its proper position. At worst, that is an additional $n-1$ comparisons. This gives is the recurrence:

$$\begin{aligned} C(n) &= C(n-1) + n - 1 \\ &= [C(n-2) + (n-2)] + (n-1) \\ &= [C(n-3) + (n-3)] + (n-2) + (n-1) \end{aligned}$$

Continue this until we can apply our stopping condition:

$$\begin{aligned} C(n) &= C(1) + 1 + 2 + ... + (n-1) \\ &= 0 + 1 + 2 + ... + (n-1) \\ &= \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2) \end{aligned}$$

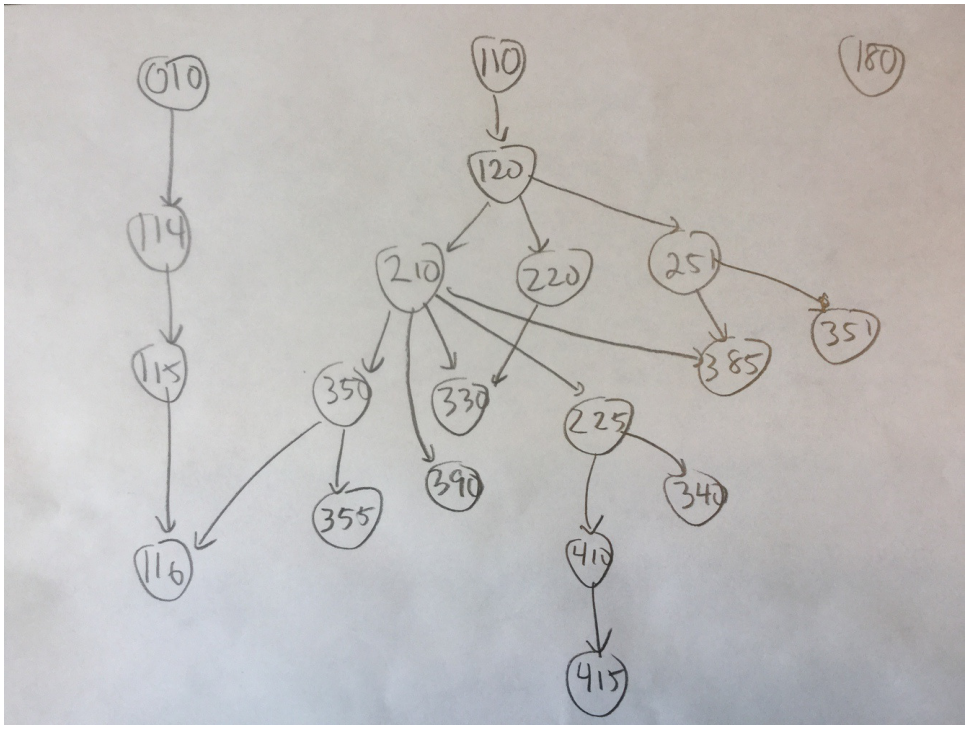
Topological Sort

Our next algorithm operates on a special class of graph structures: *directed acyclic graphs*, or *dags*.

Dags naturally arise in many problems, but we will assume that we are given a dag and wish to perform a *topological sort* of the graph vertices. A toplogical sort of a dag is an ordering of the vertices such that for every edge in the dag, the starting vertex of the edge is listed before the ending vertex.

Note that it makes no sense to attempt a topological sort if the graph is either undirected or if it has a cycle.

One example of a dag might be a course prerequisite graph.



One algorithm to compute a topological sort of a dag is based on a depth-first traversal of the graph is described in the text.

We will look at a second option (also in the text) called the *source removal algorithm*.

The algorithm proceeds as follows:

ALGORITHM SOURCEREMOVAL(G)
  //Input: a dag $G = \{V, E\}$
  $o \leftarrow$ a new empty list
  while $\exists$ a source vertex in G **do**
    $v \leftarrow$ a source vertex from G
    $o.append(v)$
    $G.remove(v)$ // also removes all incident edges
  if G is empty **then**
    return o
  else
    flag error // G was not a dag

Generating Combinatorial Objects

One of the things we saw in the "brute force" algorithm discussion was that we sometimes need to enumerate all possible solutions for a given problem.

Permutations

You already implemented one mechanism for generating all permutations of a list of items for a problem set.

The text describes two additional methods for enumerating permutations that we will not discuss in class. It is worth reading about them.

Subsets

We saw the need to generate all possible subsets (*i.e.*, a *power set*) when discussing the knapsack problem.

The text describes four approaches.

For the first, we will use a direct descrease and conquer approach. We will consider each element in the set, and generate all of the sets that do not have that element. Then, the subsets are exactly those plus the same group of subsets but with this element added in. See the table in Figure 4.10, p. 147, for an example.

A quite clever method (and your instructor's favorite mechanism) involves using an integer value, treating its bottom n bits as Boolean values indicating whether one of the elements is in the power set or not. The advantage of this is that the loop to visit all subsets becomes a `for` loop, counting from 0 to $2^n - 1$. And inside the loop, we just find the "1" bits and treat the corresponding elements as being in the set, or not, as appropriate. This generates a "lexicographic order".

Another alternative is to generate the "squashed order", where we first generate the empty set, then the singletons in order, then the two-element subsets, etc. This one is left as an exercise.

Finally, there is the Gray code ordering, where each subset in the sequence differs from the next by the addition or removal of a single element.

See the algorithm on p. 148.

Decrease-by-a-constant-factor

Binary Search

We next briefly recall the idea of a *binary search* – the efficient procedure for searching for an item in a sorted array.

A binary search is often treated as an example of a divide and conquer algorithm – our next major group of algorithms, but Levitin treats it as an example of a *decrease-by-a-constant-factor algorithm*.

Here is a nonrecursive algorithm that implements binary search:

ALGORITHM BINARYSEARCH(A, K)
  //Input: a sorted array $A[0..n-1]$
  //Input: search key K
  $l \leftarrow 0$

$r \leftarrow n - 1$
while $l \leq r$ **do**
    $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
    if $K = A[m]$ **then**
        return m
    else if $K < A[m]$ **then**
        $r \leftarrow m - 1$
    else
        $l \leftarrow m + 1$
return -1

To find an item, we look for it in the middle. If we do not find it, we look only to the left (if the target is smaller than the middle element) or to the right (if the target is larger than the middle element).

We will not look in detail at the recurrence or its solution right now, just note that this is a very fast algorithm: requiring $\Theta(\log n)$ comparisons in the worst case.

Fake Coin Problem

The text discusses a version of the *fake coin problem* where we are given a set of n coins, one of which is a fake. We know it is a fake because it is lighter than the legitimate coins. To solve the problem we are able to use a balance scale, and on each side of the scale we can put 1 or more coins to compare the weights.

Discussion:

- a naive approach: weigh pairs of coins until we find the one that's lighter than the others
- a reduce-by-constant-factor approach where we weigh equal numbers of coins (half at a time), narrowing down which half contains the fake until we find it
- a reduce-by-constant-factor approach where we weigh 3 piles of coins at a time, narrowing down which third contains the fake until we find it
- does it make sense to divide into more than 3 piles?

The text has a few additonal examples that we may encounter later.