CSCI 3110

Data Structure Augmentation and More

He

Alex Safatli

Friday, October 11, 2013

Contents

Inventing (or Augmenting) a Data Structure

Example: Maximum Subrange Sum Problem, Revisited

In the third solution we investigated for the Maximum Subrange Sum Problem, we created a data structure called p. It is fairly simple but it helped us immensely.

Example: Abelian Square Detection in Strings

An **Abelian square** is a substring of the form xx' where $|x| = |x'|$, and x is a permutation of x'. For example, the word *reappear* is an Abelian square where $x =$ reap and $x' =$ pear. Also, so is AAGCTCAGAT.

NOTE Reading for today is 33.4.

NOTE See written notes for this component. Guest lecture by Norbert Zeh was done on October 18, 2013. No direct notes, but if go to `http://web.cs.dal.ca/~nzeh/Teaching/3110` — Data Structures notes for priority search trees.

Example: Range Searching

Given a number of points in Eucledian space, want a data structure to store all points and, for any query range that matches a shape, want to be able to quickly report the points within that query range. Half-plane searching, circular searching are computationally expensive; will not talk about here.

We will talk about **orthogonal range queries**: rectangular axis-dependent shapes with ninety degrees. This is an application found in databases: give all records falling in a certain range in a certain dimension, etc. In two dimensions, we can discuss the problem, but it can be easily lifted to higher dimensions.

1D Range Searching. How does 1D range searching work? Given a real line where a number of points are present, can acquire an interval and give all points within that interval. Naively, we would have $O(n)$ time if we compare all points individually. Better, we could store them into a binary search tree and determine if a query interval is empty in $O(\lg n)$ time (**emptiness queries**).

But could we combine these two solutions? In the worst case, though, linear time is the best to hope for here; the order would be $O(\lg n + k)$ where k is the output size (the number of points to report). Let us say we have a sorted array of points where we can perform binary search to find the lower boundary (in $O(\lg n)$ time) and then one would walk down until hit a value greater than. Note that we are storing the data here and building a structure; sorting will have to be done but it has been done when creating the data structure. This will make queries extraordinarily fast. We have an $O(\lg n + k)$ solution.

Binary Search Tree. But is there something analogous that can be done on a **binary search tree**? Above is nothing but an explicit representation of what a binary search tree is. Find the smallest element no less than a by using it as a search key. To find the range, one would merely traverse the tree until find the first element outside the query range.

This is analogous to before – but what is the cost? A number of internal nodes were visited, but what is visited is only what is in-between and nothing outside. The total length of the two paths together from the root to a and b is $2h$; this is bound by $O(\lg n)$ if is a balanced binary search tree. Note that there is $\leq k$

Example: Range Searching continued on next page...

leaves.We also know that one of the subtrees off the path(s) has a certain number x leaves and therefore $\leq 2x$ nodes. But this means we bound the entire portion between the two paths by $2k$ nodes. This is $O(\lg n + k)$ time.

2D Range Searching. Can we use the data structure we considered here? Consider the query range as, rather than a four-sided query range, but as a **three-sided query range**. We know how to consider only one axis, say the x-axis. But considering the y-axis is merely a matter of giving everything above a certain value (if the top of the query range is infinite). An easy solution is to iterate over a sorted array. This contributes $O(1 + k)$ time. But a **binary heap** is much more useful here.

Binary Heap. Each node has the property that its children are less than or equal to it. Finding the maximum element and then constantly popping them is one way to do this is $O(k \lg n)$, but this is too expensive and destructive. However, one would look at $\leq 3k + 1$ nodes in constant time (if considering both children). All in one data structure, one could have a data structure that is a binary search tree for the x-coordinates and a heap for the y-coordinates. This would involve $O(numberoftrees + k)$ lookups $= O(\lg n + k)$.

This is known as a **Priority Search Tree**.

Greedy Algorithms

Example: Activity Selection

The pseudocode for `Activity_selection` is shown below:

```
   Activity_selection(s,f,n)
1  Sort both s and f by f[i] in increasing order
2  A <-- {1}
3  k <-- 1
4  for i <-- 2 to n do
5    if S[i] >= f[k] then
6      A <-- A U {i}
7      k <-- i
8  return A
```

This is $O(n \lg n)$.

The Knapsack Problem

Let us say we have a problem where we have n items. Each respective `Item` has a weight w_i in kg and a value v_i in dollars. We also have a **knapsack** which has a maximum weight capacity W. The problem here is that we wish to choose a set of items in order to maximize the value being held in the knapsack.

Example. Here is an example scenario. Consider a $W = 100, n = 6$. Items $1, 2, 3, 4, 5, 6$ have, respectively, values and weights:

item	1	2	3	4	5	6
value	80	70	85	40	75	65
weight	25	40	70	15	20	5
value/w	3.2	1.75	1.21	2.67	3.75	13

The Knapsack Problem continued on next page...

Possible (greedy) *strategies* here involve:

1. Most valuable item first,
2. Heaviest items first,
3. Lightest items first, and
4. Item with highest $\frac{\text{value}}{\text{weight}}$ ratio first.

These *all fail*. The optimal solution is to carry $\{1, 2, 5, 6\}$, altogether possessing a weight of 90 kg and a value of 290 dollars. Strategy 1 would choose $\{3, 1, 6\}$ (weight 100 kg and value 230 dollars), Strategy 2 would choose the same, Strategy 3 would choose $\{6, 4, 5, 1\}$ (weight 90 kg and value 260 dollars), and Strategy 4 would choose $\{6, 5, 1, 4\}$.

Note that *this problem cannot be solved using greedy algorithms*. It can, however, be solved using **dynamic programming** (see later). One variant, also, can be solved using a **greedy algorithm**.

This variant is known as the **Fractional Knapsack Problem**.

Example: Fractional Knapsack Problem

A variant of the **Knapsack Problem** where where each item is a sack of substance that can be arbitrarily divided (e.g., a bag of sugar, salt, gold dust).

The strategy above that considers ratios (4) would work here: considering the same example, the set of items $\{6, 5, 1, 4, \text{a } \frac{35}{40} \text{ portion of } 2\}$ would be chosen with a weight 100 kg and a total value of 321.25 dollars.

Consider the following pseudocode to solve this problem.

```
   greed(n,v,w,W)
1  Sort both v and w by the ratio v[i]/w[i] in decreasing order.
2  free <-- W // Start with full capacity.
3  sum  <-- 0 // Total value of items chosen.
4  for i <-- 1 to n do
5    x <-- min(w[i],free) // Ensure is capped by W.
6    sum <-- sum + v[i] * (x/w[i]) // Compute value.
7    free <-- free - x
8  return sum
```

This intuitively works. Let us consider the *correction analysis.* This is the general structure for the proof of a greedy algorithm.

Proof. By contradiction.

First consider the case in which `v[i]/w[i]` are all distinct. Assume to the contrary that the optimal solution S is better than our greedy solution G. Then S must agree for some number of items (perhaps 0) with G and then differ. Assume that the items have been ordered by `v[i]/w[i]` in *decreasing order*: let k be the first item of which S and G choose a different amount. Since G chooses the maximum possible amount of an item k that can fit in the knapsack with the previously chosen item, then S must choose less of k than G does.

Say that G chooses g kg of k and S chooses s kg of k. We can exchange $g - s$ kg of later items that S chooses with $g - s$ kg of item k. Doing so would only *increase* the total value of S since later items have lower ratios. This is a contradiction: this is a solution even better than S (S is not optimal).

For the more general case in which `v[i]/w[i]` are not all necessarily distinct, can group the items with the same ratio into the same group, applying similar reasoning. □

NOTE Reading for today is 16.2, and for next lecture is 15.2.

Example: Egyption Fractions

Egyptian Fractions are expressions of a certain form $\frac{1}{x_1} + \frac{1}{x_2} + ... + \frac{1}{x_n}$ where x_i is a distinct positive integer; this is otherwise known as a sum of **unit fractions**. For example, $5/6 = 1/2 + 1/3$ and $47/72 = 1/2+1/9+1/24$; think about the latter as a distribution of 47 sacks of grain to 72 people fairly – $36 = 72\times1/2$, $8 = 72 \times 1/9$ and $3 = 72 \times 1/24$.

Greedy (Fibonacci, 1202). Proposed that any fraction $\frac{x}{y}$ can be represented as:

$$\frac{x}{y} = \frac{1}{\lceil \frac{y}{x} \rceil} + \frac{(-y) \mod x}{y \times \lceil \frac{y}{x} \rceil} \tag{1}$$

Realize that $a \mod n = a - n\lfloor \frac{a}{n} \rfloor$, which is equation 3.8 on page 54, and that $\lceil x \rceil = -\lfloor -x \rfloor$. Each time we apply this, the numerator of the second term decreases. Using this, we find that: $47/72 = 1/2 + 1/7 + 1/101 + 1/50904$ which is not the shortest representation and does not minimize x_n.

Another example is the greedy expansion of 31/311.

An open problem is one where we consider that every fraction of the form $\frac{4}{n}$, where n is an odd integer, has an Egyptian fraction representation with 3 terms.

Dynamic Programming

Key Concepts

Dynamic Programming is typically applied to optimization problems. It also typically reduces complexity drastically, from 2^n to $O(n^3)$ or $O(n^2)$ or even $O(n)$. The key principle is to *avoid enumerating all possibilities.*

Example: Matrix-Chain Multiplication

Considering the matrices involved in multiplication, we consider a **matrix** $A = (a_{ik})$ which is an $m \times n$ matrix and $B = (b_{kj})$ which is an $n \times p$ matrix. The number of columns has to equal to the number of rows of the second matrix for two matrices to be multiplied together.

Let us consider the matrix $A \times B = C$ where $C = (c_{ij})$ which is a $m \times p$ matrix where $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. Realize that the number of scalar multiplications is equal to mnp for an entire pairwise matrix multiplication.

Multiplying three matrices is also imaginable. Consider the following scenario:

```
matrix    dimension
M_1       10 x 100
M_2       100 x 5
M_3       5 x 50
```

Example: Matrix-Chain Multiplication continued on next page...

Computing $M_1 \times M_2 \times M_3$ is equivalent to $M_1(M_2M_3)$ or $(M_1M_2)M_3$. The former has $100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$ multiplications. The latter has $10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$ multiplications. The difference here is a factor of 10.

Problem. We can define the following problem from this. Given an input chain of n matrices $(M_1, M_2, ..., M_n)$ where M_i has dimension $p_{i-1} \times p_i$ (rows $\times$ columns). The goal here is to fully parenthesize the product $M_1M_2...M_n$ in a way that minimizes the total number of scalar multiplications.

Brute-Force. Without dynamic programming, checking all possibilities in a brute-force manner has a respective running time. Suppose $W(n)$ is the number of ways to multiple n matrices. Observe that the last multiplication is either $M_1(M_2...M_n)$, $(M_1M_2)(M_3...M_n)$, ..., or $(M_1M_2...M_{n-1})M_n$ — there are $n-1$ possibilities here.

We can express the number of ways to multiply these matrices in a recursive fashion. Realize that computing the first k matrices means there are $W(k)$ different ways; the remaining matrices would have $W(n-k)$ different ways.

$$W(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} W(k)W(n-k) & n \neq 1 \end{cases} \tag{2}$$

Provable by induction, we can define this function by the following.

$$W(n) = \frac{\binom{2n-2}{n-1}}{n} \tag{3}$$

Where, recall, we know that $\binom{a}{b} = \frac{a!}{b!(a-b)!}$ and Stirling's Approximation states that $n! \approx n^n e^{-n}\sqrt{2\pi n}(1 + \Theta(1/n))$. We can then approximate:

$$\binom{2n}{n} = \frac{(2n)!}{(n!)^2} \approx \frac{(2n)^{2n}\sqrt{2\pi 2n}}{n^{2n}2\pi n} = \frac{2^{2n}}{\sqrt{\pi n}} \tag{4}$$

In other words, $W(n) \approx \frac{4^n}{n^{3/2}}$ which is exponential.

How do we recover from this? Realize that **Dynamic Programming** avoids *all of the recomputation of the recursive algorithm.*

Dynamic Programming Solution. Consider the following observation.

1. Suppose the optimal method to compute $M_1M_2...M_n$ were to first compute $M_1M_2...M_k$ (in some order), then computes $M_{k+1}...M_n$ (in some order), and multiplies these two together.
2. Then, the method to compute $M_1M_2...M_k$ *must be optimal*, for otherwise, we could substitute a superior method and improve the optimal method.
3. We can say the same for $M_{k+1}...M_n$; it must also be optimal.
4. To find the best k, there are $n-1$ possibilities.

The method goes like this. Let $m[i,j]$ be an array value possessing the optimal cost for computing $M_iM_{i+!}...M_j$. Note we can consider two subproducts $Mi...M_k$ and $M_{k+1}...M_j$, each with respective costs. We can define this as:

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} m[i,j] + m[k+1,j] + p[i-1] \times p[k] \times p[j] & i \neq j \end{cases} \tag{5}$$

Note that equation 7 can be converted directly into a recursive algorithm. But this is NOT AN EFFICIENT SOLUTION — it is a very bad solution and is a common mistake for those starting dynamic programming.

```
   Bad_Recursive_Mult(i,j)
1  if i = j then
2    return 0
3  else
4    minCost <-- infinity
5    for k <-- i to j-1 do
6      a <-- Bad_Recursive_Mult(i,k)
7      b <-- Bad_Recursive_Mult(k+1,j)
8      minCost <-- min(minCost,a+b+p[i-1]*p[k]*p[j])
9  return minCost
```

Realize that this is *exponential* and just as bad as the brute force algorithm.

The key of writing the pseudocode for the dynamic programming function is to compute the subproblems in the *correct* order.

1. compute $m[i, i]$ for all i,
2. compute $m[i, i + 1]$ for all i,
3. compute $m[i, i + 2]$ for all i,
4. and so on.

ACTUAL SOLUTION. Let's look at the proper solution now.

```
   Matrix_Multi_Order(p,n)
1  for i <-- 1 to n do
2    m[i,i] <-- 0
3  for d <-- 1 to n-1 do // computing m[i,i+d]
4    for i <-- 1 to n-d do
5      j <-- i+d
6      m[i,j] <-- infinity
7      for k <-- i to j-1 do
8        // Realize that the m values have been computed already.
9        // m[i,k] --> d = k-1 < j+i
10       // m[k+1,j] --> d = j-(k+1} < j-i
11       q <-- m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j]
12       if q < m[i,j] then
13         m[i,j] <-- q
14         s[i,j] <-- k // keep tracks of choice of k for each subproblem
15 return (m,s)
```

The optimal cost should be $m[1, n]$; the running time of this function is $O(n^3)$.

Consider an example where:

```
matrix    M_1    M_2    M_3    M_4    M_5    M_6
dimens    30x35  35x15  15x5   5x10   10x20  20x25
```

For example:

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p[1]p[2]p[5] = & 1300 \\ m[2,3] + m[4,5] + p[1]p[3]p[5] = & 7125 \\ m[2,4] + m[5,5] + p[1]p[4]p[5] = & 11375 \end{cases} \quad (6)$$

We would have the following tables:

```
m   1     2     3     4     5     6
1   0     15750 7875  935   11875 15125
2         0     2625  4375  7125  10500
3               0     750   2500  5375
4                     0     1000  3500
5                           0     5000
6                                 0
i
```

```
s   1     2     3     4     5     6
1         1     1     3     3     3
2               2     3     3     3
3                     3     3     3
4                           4     5
5                                 5
6
i
```

How do we optimally *multiply the matrices*? We use recursion. To compute $M_i...M_j$ and we know k, we know the optimal order of multiplication: $(M_1...M_k)(M_{k+1}...M_j)$, and recurse on both of the two brackets, multiplying both results together. Note that the optimal k is located in $s[i,j]$.

```
   Matrix_Mult(M,s,i,j)
1  if j > i then
2    X <-- Matrix_Mult(M,s,i,s[i,j])
3    Y <-- Matrix_Mult(M,s,s[i,j]+1,j)
4    return X * Y // 3-level loop operation
5  else return M[i]
```

The running time for this operation is $T(n) = O(n)$ recursive calls+time for actual n-1 matrix multiplications. Or in other words, $T(n) = O(n) + O(m[1,n])$.

Memoization. A second approach. Recall **memoization** from the Fibonacci algorithm example.

```
   Memoized_Matrix_Mult(p,n)
   // Wrapper Function
1  for i <-- 1 to n do
2    for j <-- 1 to n do
3      m[i,j] <-- infinity
4  return Lookup(p,1,n)
```

By using this function, we can initialize all values in the array m to ∞. We then call a lookup function on this table.

```
    Lookup(p,i,j)
1   if m[i,j] < infinity then
```

```
2     return m[i,j]
3   if m = j then
4     m[i,j] <-- 0
5   else
6     for k <-- i to j-1 do
7       q <-- Lookup(p,i,k) + Lookup(p,k+1,j) + p[i-1]*p[k]*p[j]
8       if q < m[i,j] then
9         m[i,j] <-- q
10        s[i,j] <-- k
11  return m[i,j]
```

The initialization function's loops have $O(n^2)$. The `Lookup(p,1,n)` call has two cases:

1. Type 1 calls execute lines 1 and 2. The number of calls of this type, which are made as recursive calls by Type 2 calls, are $O(n^3)$; lines 1 and 2 are executed this many times.
2. Type 2 calls execute lines 1 and 3-11. The number of calls of this type are $O(n^2)$ (the number of entries equal to ∞). Lines 3-10 therefore have running time $O(n^3)$, when compounding the number of calls of that loop.

The running time of this approach is, in total, $O(n^3)$.

Example: Longest Common Subsequence

This problem has, as input, two strings x and y, each respectively of length m and n. The task here is to find the longest sequence that appears (not necessarily consecutively) in both x and y. For example, if x was "algorithm" and y was the word "exploration", the **longest common subsequence** would be the string "lort".

Brute-Force Solution. The number of possible subsequences of $x[1..m]$ is 2^m. We can tell right away that this solution is *not efficient*.

NOTE The reading for this lecture is 15.3. For next lecture, it is 15.4.

Dynamic Programming Solution. We first observe that we can look at the last character of either x or y.

1. If $x[m] = y[n] = a$, then any longest common substring or subsequence z of length k of x and y will end in $z[k] = a$. – otherwise, we could append a to z and acquire an even longer common subsequence.
2. If $x[m] \neq y[n]$, then if $z[k] \neq x[m]$, we can observe that $z[1..k]$ is a longest common subsequence of $x[1..m-1]$ and $y[1..n]$.
3. By this same logic, if $x[m] \neq y[n]$ and $z[k] \neq y[n]$, we can observe that $z[1..k]$ is a longest common subsequence of $x[1..m$ and $y[1..n-1]$.

We can also establish an array $c[i,j]$ which contains the length of the longest common subsequence of $x[1..i]$ and $y[1..j]$.

$$c[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & x[i] = y[j] \\ \max(C[i-1,j], C[i,j-1]) & x[i] \neq y[j] \end{cases} \tag{7}$$

Example: Longest Common Subsequence continued on next page...

Consider the following pseudocode.

```
    LCS_Length(x[1..m],y[1..n])
1   for i <-- 1 to m do
2     c[i,0] <-- 0
3   for j <-- 1 to n do
4     c[0,j] <-- 0
5   for i <-- 1 to m do
6     for j <-- 1 to n do
7       if x[i] = y[j] then
8         c[i,j] <-- c[i-1,j-1] + 1
9       else
10        c[i,j] <-- max(c[i-1,j],c[i,j-1])
11  return c[m,n]
```

As an exercise, consider the above example of the two words "exploration" and "algorithm"; fill in the table for the lengths of the longest common subsequences.

The running time of this algorithm is $O(mn)$.

How do we compute the string itself?

```
   LVS(i,j)
1  if i = 0 or j = 0 then
2    return empty string
3  else if c[i,j] > max(c[i-1,j],c[i,j-1]) then
4    return LCS(i-1,j-1) + x[i] // Case 1
5  else if c[i-1,j] >= c[i,j-1] then
6    return LCS(i-1,j) // Case 2
7  else
8    return LCS(i,j-1) // Case 3
```

The running time for computing the string itself (excluding the previous step) would be $O(m+n)$. Each call reduces i, j, or both.

Example: Making Change

Given an amount of money in cents, we wish to make change using a system of denomination, using the smallest number of coins possible to make change.

In the Canadian coinage system, we have, if we include pennies, we can consider an algorithm where the input is an integer n, an amount $0 \leq n < 500$ where, if the change is 500 cents, we can just use a 5 dollar bill. the task here is to express $n = 200a + 100b + 25c + 10d + 5e + f$; these are the number of coins for each value used — they are integers ≥ 0. This expression must have a minimized sum; $a+b+c+d+e+f$ is minimized.

Greedy Algorithm. See Software Development Assignment. Choose as many toonies as possible, then for $n - 200a$, choose as many loonies as possible, and so on. For example, we have change 4.87 dollars where $a = 2, b = 0, c = 3, d = 2, e = 0, f = 2$. Total number of coins here is 8. Realize that this algorithm is not always optimal; for example, using a strange system of domination where we have coins (12, 5, 1), the greedy algorithm would assign 1 coin of value 12 and 3 coins of value 1.

Example: Making Change continued on next page...

NOTE Reading for today is 15.4. Content for Midterm II ends here.

Dynamic Programming Solution. Have input $n > 0$, array $d[1..k]$ (face values or denominations of the coins; assume $d[1] = 1$ and $d[i] < d[i+1]$). Observe that the optimal solution must use *coins* and therefore some denomination $d[i]$. Optimally, this would be the optimal solution for $n - d[i]$ and then adding one coin of denomination $d[i]$.

$$opt[j] = \begin{cases} 1 & j = d[i] \text{ for some } i \\ 1 + \min_{\forall i, d[i] < j}(opt[j - d[i]])) & \text{otherwise} \end{cases} \tag{8}$$

We can therefore define a recursive formula where all i are considered to try all $d[i]$ such that $d[i] \leq n$. Consider the following pseudocode where `opt[j]` is the optimal number of coins to make change for j cents. We compute `opt[j]` for $j = 1, 2, 3, ..., n$.

```
    coins(n, d[1..k])
1   for j <-- 1 to n do
2     opt[j] <-- infinity
3   for i <-- k downto 1 do
4     if d[i] = j then
5       opt[j] <-- 1
6       // Store largest face value.
7       largest[j] <-- d[i]
8     else if d[i] < j then
9       // Use second case of formula for opt.
10      a <-- 1 + opt[j-d[i]]
11      if a < opt[j] then
12        opt[j] <-- a
13        largest[j] <-- d[i]
14  return opt[n]
```

For example, given $d = \{1, 4, 5\}$, and $n = 8$, we would acquire a solution of $8 = 5 + 1 \times 3$ for the Greedy Solution. For the dynamic programming solution, we would acquire the following.

j	`opt[j]`	`largest[j]`
1	1	1
2	2	1
3	3	1
4	1	4
5	1	5
6	2	5
7	3	5
8	2	4

Table 1: Solution for $d = \{1, 4, 5\}$ using dynamic programming.

Realize that $opt[8] = 1 + min \begin{cases} opt[8-5] = 3 \\ opt[8-4] = 1 \\ opt[8-1] = 3 \end{cases} = 2.$

From this example, we can identify that 8 has a representation of 2 (it can be represented by the sum of

two coins), and the largest coin used has face value of 4. To make change for $8 - 4 = 4$ cents, we can see by $opt[4]$ this can be expressed using one coin of size 4. The remaining value is 0 and we have made change completely for the given n. Therefore, $8 = 4 + 4$ using the dynamic solution; this process of finding the end representation of n can be done recursively. To make the change, consider the following pseudocode.

```
   make_change(n,largest)
1  C <-- empty set // multiset
2  while n > 0 do
3    C <-- C in union with {largest[n]}
4    n <-- n - largest[n]
5  return C
```