Data Structures Homework #6

Ari Feiglin

Question: 6.1:

Given an array of natural numbers $A[1 \dots n]$ and a number k, we want to know if there exists a subarray (not necessarily continuous) of A whose sum is exactly k.

(1) The first step is to come up with a naive solution to the problem. Such a naive solution is simple: we iterate over every possible subarray of A and check if its sum is k. Unfortunately, there are 2^n such subarrays, so this takes $\Omega(2^n)$ time.

(2) Now let's come up with a recursive formula for this problem. Let $f(i,s)$ give a set of indexes $I \subseteq \{1,\dots,i\}$ such that $\sum_{i \in I} A[i] = s$. If there doesn't exist such a subarray, $f(i,s) = \varnothing$. Thus know that if $i = 0$, then $f(i,s) = \varnothing$. If $A[i] = s$, then we can return $\{i\}$. And if $A[i] > s$, i can't be part of the indexing set, so $f(i,s) = f(i-1,s)$ since we can ignore i. If $f(i-1, s-A[i]) \neq \varnothing$, then $f(i,s) = f(i-1,s-A[i]) \cup \{i\}$ since the sum over the indexes in $f(i-1,s-A[i])$ yields $s - A[i]$, so adding $A[i]$ gives s, as required. Lastly, if $f(i-1,s-A[i]) = \varnothing$, then i can't be in the indexing set since there exists no indexing set in $[i-1]$ whose sum is $s - A[i]$, so $f(s,i) = f(s,i-1)$.

So all in all, we have that:

$$f(s,i) = \begin{cases} \varnothing & i = 0 \\ \{i\} & A[i] = s \\ f(i-1,s) & A[i] > s \\ f(i-1,s-A[i]) \cup \{i\} & f(i-1,s-A[i]) \neq \varnothing \\ f(i-1,s) & \text{else} \end{cases}$$

We want to compute $f(n,k)$ (a set $I \subseteq \{1,\dots n\}$ whose sum is k).

Unfortunately, in the worst case we need to check $f(i-1,s-A[i])$ and $f(i-1,s)$. This means that in the worst case $\mathrm{T}(n) = 2\,\mathrm{T}(n)$, which means that $\mathrm{T} \in \Theta(2^n)$, which is not of much help.

(3) But notice that if we define an array $B[0 \dots n, 1 \dots k]$ such that $B[i,s] = f(i,s)$ we can fill the array from the bottom up to get $B[n,k]$. So B will be a two-dimensional table where each node will contain a linked list (for constant time insertion) which equals $f(i,s)$.

We begin filling up B from $B[0, 1 \dots k]$ and we set each node to an empty linked list. Then we iterate over i from 1 to n and iterating inside on s from 1 to k, for every $B[i,s]$ we set it as follows:

- If $A[i] = s$, then we set $B[i,s]$ to a linked list with a single node i.
- If $A[i] > s$, then we set $B[i,s]$ to $B[i-1,s]$.
- If $B[i-1, s-A[i]]$ is not an empty linked list, then we set $B[i,s]$ to a node of value i with a pointer to $B[i-1,s-A[i]]$ (thus adding a node to the linked list).
- Otherwise, set $B[i,s]$ to $B[i-1,s]$.

Each step here takes constant time, and since the dimension of B is $(n+1) \times k$, it takes $\mathcal{O}(nk)$ time to fill B. Furthermore, each node takes a constant amount of space (since they're just nodes in a linked list), so the array B takes $\mathcal{O}(nk)$ space. Once B is filled, $B[n,k]$ will equal $f(n,k)$, our goal.

This algorithm can be written with the following pseudocode:

Input : An array, $A[1, \ldots, n]$, of naturals and a natural number k.
Output: A linked list of indexes of A whose sum is k.

```
1  for s ← 1 to k do
2  |  B[0, s] ← new Node();
3  end
4  for i ← 1 to n do
5  |  for s ← 1 to k do
6  |  |  if A[i] = s then B[i, s] ← new Node(i);
7  |  |  else if A[i] > s then B[i, s] ← B[i − 1, s];
8  |  |  else if B[i − 1, s − A[i]] ≠ ∅ then B[i, s] ← new Node(i, B[i − 1, s − A[i]]);
9  |  |  else B[i, s] ← B[i − 1, s];
10 |  end
11 end
12 return B[n, k];
```

Question: 6.2:

Given k different types of coins with values $v_1 < \cdots < v_k \in \mathbb{N}_1$ and a sum to pay n, create an algorithm to find the least amount of coins needed to pay *exactly* n.

(1) The naive solution to this problem is to go over every possible combination of coins, and then iterate over every possible amount of each of those coins where the sum is n and find the minimum. This has a time complexity of $\Omega(2^n)$ (in fact it may be even worse as we need to iterate over the amounts of the coins as well), which is not ideal.

(2) Let $f(m)$ equal the minimum number of coins needed to pay m. So we want to compute $f(n)$.

If $m = 0$, then $f(m) = 0$ since it takes no coins to pay 0. Otherwise, $f(m) = \min\{1 + f(m - v_i) \mid i \in [k], v_i \leq m\}$ since $1 + f(m - v_i)$ is the minimum number of coins needed to pay m if we use coin v_i, so we find the minimum for this out of all is. Note that v_i must be less than or equal to m in order to be used to pay for m.

Thus we have the recurrence:

$$f(m) = \begin{cases} 0 & m = 0 \\ \min\{1 + f(m - v_i) \mid i \in [k], v_i \leq m\} & \text{else} \end{cases}$$

But computing this has a time complexity of:

$$T(n) = \sum_{i=1}^{k} T(n - v_i) + \mathcal{O}(1)$$

Which has a maximum (worst case) if $v_i = i$ (since this is the minimum value of v_i):

$$T(n) = \sum_{i=1}^{k} T(n - i) + \mathcal{O}(1)$$

But this is exponential ($T(n) \geq 2T(n-2) + \mathcal{O}(1) \in \Omega\left(\sqrt{2}^n\right)$), which is not good.

(3) Instead, let us define $B[0 \ldots n]$ to be an array where $B[m] = f(m)$. We start filling B from index 0 to n. First we set $B[0] = 0$ and then for every m we iterate over i from 1 to k, check if $v_i \leq m$ and compute the minimum of $1 + B[m - v_i]$ (as this is the minimum of $1 + f(m - v_i)$). We then return $B[n]$ as that is $f(n)$.

In pseudocode, the algorithm is:

Input : A set of coin values $\{v_1, \ldots, v_k\}$ and a number n.
Output: The minimum number of coins needed to pay exactly n.

```
1 B[0] ← 0;
2 for i ← 1 to n do
3 |   B[i] ← min{1 + B[m − v_i] | i = 1 ... k if v_i ≤ m};
4 end
5 return B[n];
```

Since every iteration takes $\mathcal{O}(k)$ time, the algorithm takes $\mathcal{O}(nm)$ time in total. And the size of B is n, so it takes $\mathcal{O}(n)$ space.

Question: 6.3:

We are given a function T such that for every string S and two indexes $i \leq j$, $T(S, i, j)$ returns whether or not $S[i \dots j]$ is a valid word. We want to create an algorithm where given a string S it will determine if there is a partition of S into valid words.

Suppose we're given a string S with length n.

(1) The naive method would be to just go over every possible partition of S and then check if it is a valid partition in this context. There are 2^n possible partitions of S (each partition is uniquely characterized by the end indexes of each subset, so we can construct a bijection from the powerset of $[n]$ to the set of partitions of S). This means that this method has a time complexity of $\Omega\left(2^n\right)$, which is not good.

(2) Let $f(S, i)$ denote whether there is a valid partition of $S[i \dots n]$. So we want to compute $f(S, 1)$.

If $i \leq n$, then $f(S, i) = 1$ (true) if there exists some $i \leq k \leq n$ such that $S[i \dots k]$ is a word, and $f(S, k+1) = 1$, as this corresponds to a partition where $S[i \dots k]$ is part of the partition. If there is no k like this, then $f(S, i) = 0$ since for every word starting at i, it cannot form a partition, so there cannot be any partition. If $i = n+1$ then we'll define that $f(S, i) = 1$. This is because the empty word should count as a valid partition (vaccuously), and also because this recurrence will eventually lead to $i = n+1$, and in such a case, that means that before i there is a valid partition and this partitions all of S.

So:

$$f(S, i) = \begin{cases} 1 & i = n+1 \\ 1 & \exists i \leq k \leq n : T(S, i, k) \text{ and } f(S, k+1) \\ 0 & \text{else} \end{cases}$$

The time complexity of naively computing this recurrence is:

$$T(n) = \sum_{i=1}^{n} T(n-i)$$

Which is exponential.

(3) We define an array $B[1 \dots n+1]$ such that $B[i] = f(S, i)$. So we want to find $B[1]$, as this is equal to $f(S, 1)$.

We fill it up from the end backwards like so: first we set $B[n+1] = 1$, and then for every i, we iterate over k from i to n and check if $T(S, i, k) = 1$ and $f(S, k+1) = 1$, and if so, set $B[i] = 1$. Otherwise, set $B[i] = 0$.

Pseudo code for this algorithm:

```
Input  : A string S.
Output: Whether or not there's a partition of S into valid words.
1 B[n + 1] ← 1;
2 for i ← n to 1 do
3 |   for k ← i to n do
4 |   |   if T(S, i, k) = 1 and B[k + 1] = 1 then
5 |   |   |   B[i] ← 1;
6 |   |   end
7 |   end
8 end
9 return B[n];
```

This has a time complexity of:

$$\sum_{i=1}^{n} \sum_{k=n-i+1}^{n} \mathcal{O}(1) = \sum_{i=1}^{n} \mathcal{O}(i) = \mathcal{O}\left(n^2\right)$$

And B takes $\mathcal{O}(n)$ space, so the algorithm has a space complexity of $\mathcal{O}(n)$.

Question: 6.4:

Given an algebraic expression with natural numbers using only addition and subtraction, determine the optimal place to place parentheses (you cannot split a number with parentheses, eg. 2(3)) in order to maximize the value of the expression.

(1) The naive approach is to check every possibile way of writing out parentheses, and compute the maximum. There are an exponential number of such ways (we can define a recurrence $a_n = \sum_{i=1}^{n} a_i + a_{n-i}$, this is a lower bound to the number of ways to add parentheses since we add parentheses around the first i values, and then around the next $n-i$ values. This recurrence is obviously exponential at least). So this has a lower bound time complexity of $\Omega(2^n)$.

(2) First let's format our input. Let a_i be the ith number, and $\circ_i$ be the ith operator (so $\circ_i \in \{+, -\}$). That means that the expression is equal to:

$$a_1 \circ_1 a_2 \circ_2 \cdots \circ_{n-2} a_{n-1} \circ_{n-1} a_n$$

Let's define $M(i,j)$ to be the maximum value of the expression between indexes i and j, that is the maximum value of:

$$a_i \circ_i a_{i+1} \circ_{i+1} \cdots \circ_{j-1} a_j$$

And similarly, let $m(i,j)$ be the minimum of this expression. We thus want to compue $M(1,n)$. Notice then that $M(i,i) = m(i,i) = a_i$, since the maximum and minimum value of the expression a_i is, in fact, a_i.

Let's first quickly prove that for every expression with a length greater than one, it can be rewritten as an equivalent expression where on every level, there are two groups of parentheses. We will prove this by induction, starting on an expression of length 2, which has the form $a \circ b$. This is obviously equal to $(a) \circ (b)$, and we are finished. Now for the inductive step, suppose we have an expression $e_1 \circ_1 e_2$. Note that e_1 and e_2 themselves may include parentheses, but for explicity, suppose that e_1 is in fact totally wrapped in parentheses, so I'll write $(e_1) \circ_1 e_2$. I can do this since all operations have the same precedence, so in general $e_1 \circ e_2 = (e_1) \circ e_2$. I cannot do this to e_2 since $\circ_1$ may be a minus, and $-1 + 2 \neq -(1+2)$. Now suppose that $e_2 = (\tilde{e}_2) \circ_2 e_3$. Note that $\tilde{e}_2$ may be a single number, in which case $\tilde{e}_2 = (\tilde{e}_2)$, but $\tilde{e}_2$ is not an empty expression, while e_3 may be. Then we have the expression

$$(e_1) \circ_1 (\tilde{e}_2) \circ_2 e_3$$

This is equal to $((e_1) \circ_1 (\tilde{e}_2)) \circ_2 e_3$. By our inductive hypothesis, we can assume that e_1 has two parentheses on every level. Now, we know that $e_3 = (\tilde{e}_3) \circ_3 e_4$, and we can recursively add this to the parentheses like we did with $\tilde{e}_2$, and in the end we'll be left with something in the form:

$$(e) \circ (\tilde{e}_n)$$

As required.

This is important since it means that when computing the maximum or minimum of the expression, we can place only two parentheses at a time and this is ensured to give us a result (so we don't need to worry about placing parentheses like $(e_1) \circ_1 (e_2) \circ_2 (e_3) \circ_3 \ldots$, we can be satisfied with $(e_1) \circ (e_2)$).

So in order to compute $M(i,j)$ we can iterate over k from i to $j-1$ and break up the expression there. That is, we split it up into $(e_{i,k}) \circ_k (e_{k+1,j})$. The maximum value of this is $M(i,k) + M(k+1,j)$ if $\circ_k = +$ and $M(i,k) - m(k+1,j)$ if $\circ_k = -$. This is just equal to $\max\{M(i,k) \circ_k M(k+1,j), M(i,k) \circ_k m(k+1,j)\}$. Therefore:

$$M(i,j) = \begin{cases} a_i & i = j \\ \max\limits_{i \leq k < j} \max\left\{ M(i,k) \circ_k M(k+1,j), M(i,k) \circ_k m(k+1,j) \right\} & i \neq j \end{cases}$$

Similarly:

$$m(i,j) = \begin{cases} a_i & i = j \\ \min\limits_{i \leq k < j} \min\left\{ m(i,k) \circ_k m(k+1,j)\ ,\ m(i,k) \circ_k M(k+1,j) \right\} & i \neq j \end{cases}$$

This has a time complexity of:

$$T(n) = \sum_{k=1}^{n-1} T(k) + T(n-k) = 2\sum_{k=1}^{n-1} T(k)$$

This is obviously at least exponential, and is therefore not good by itself.

(3) As always, we can use dynamic programming. We define two arrays, the $M[1 \dots n, 1 \dots n]$ array, and the $m[1 \dots n, 1 \dots n]$ array. $M[i,j]$ will equal $M(i,j)$ and $m[i,j]$ $m(i,j)$. We begin by setting $M[i,i], m[i,i] \leftarrow a_i$. We start by iterating from $i \leftarrow n$ to 1 and $j \leftarrow i-1$ to n (since $i \leq j$ and $M(i,j)$ relies on $M(i',j')$ where $i \leq i'$ and $j \geq j'$) and we set $M[i,j]$ and $m(i,j)$ according to the reccurrences defined above.

Setting each node takes $\mathcal{O}(n)$ time, and there are n^2 nodes, so all in all this algorithm has a time complexity of $\mathcal{O}(n^3)$. Both M and m take n^2 space, so the space complexity is $\mathcal{O}(n^2)$.