Hannah Roach

CSC 388: Programming Languages

**Description of Algorithm and Complexity**

At a high level, the **has-subtarget?** algorithm accepts a list, performs a check, and returns a Boolean answer. The **has-subtarget?** algorithm uses recursion to check for two distinct sub-lists whose sums are equal; let’s call these values **x1** and **x2.** It then checks if **x1** and **x2** equal a value not in either of these two lists; let’s call this the subtarget, **x3**. If the algorithm finds a subtarget, **x3**, equal to **x1** and **x3**, the algorithm returns true (**#t**), otherwise the algorithm returns false (**#f**).

1. First, the algorithm accepts a list, **List,** and six parameters, **x1**, **x2**, **x3**, **y1**, **y2**, and **y3.**

**(define has- subtarget? (lambda (List x1 x2 x3 y1 y2 y3)**

1. In each test, these values are initialized to zero. The function then checks if the list is empty.

**(null? List)**

1. If the list is not empty, the algorithm performs a set of recursive calls. In the lines below, you can see that the algorithm passes the remainder of the list, **(cdr List)**, into the function.

**(or**

**(has- subtarget? (cdr List) x1 (+ (car List) x2) x3 (+ (car List) x2) y2 y3)**

**(has- subtarget? (cdr List) (+ (car List) x1) x2 x3 y1 (+ (car List) x1) y3)**

**(has-subtarget? (cdr List) x1 x2 (car List) y1 y2 (car List))**

**)**

The first recursive call, **(has- subtarget? (cdr List) x1 (+ (car List) x2) x3 (+ (car List) x2) y2 y3),** sets the second and fourth parameter to **(+ (car List) x2).** This adds the first value of the list to **x2**.

The second recursive call, **(has- subtarget? (cdr List) (+ (car List) x1) x2 x3 y1 (+ (car List) x1) y3),** sets the first and fifth parameter to the same value, **(+ (car List) x1)**.

The third recursive call, **(has-subtarget? (cdr List) x1 x2 (car List) y1 y2 (car List))**, sets the third and sixth parameter to the first value of the list, **(car List)**.

The first and second recursive call find the sum of two sub-lists. The third recursive call finds a value not in either of the sub-lists.

If you were to create a call table of these recursive calls, it would look like the following:

The first node is where the function is called. Each subsequent level multiplies the number of nodes by the number of recursive calls. The number of levels is determined by the number of elements in the list. So, if there were five elements in a list, there would be a total of 1 + 1\*3 + 3\*3 + 4\*3 + 5\*3, recursive calls. This can be represented as

where **L** is the number of elements in the list and **n** is the range of numbers between 0 and **L**. In terms of Big Oh complexity, this can be represented as

BigO(

1. When the list **(cdr List)** is null, the list has been completely iterated through and **(null? List)** returns true. The algorithm then checks for equivalency between the sub-lists, **(= x1 y1)** and **(= x2 y2)**, and the sub-target, **x3** (see below). If all conditions return true, the algorithm returns #t. If the conditions do not return true, the algorithm returns #f.

**(define has-subtarget? (lambda (List x1 x2 x3 y1 y2 y3)**

**(cond**

**[(null? List)**

**(and**

**(= x1 y1)**

**(= x2 y2)**

**(= x3 x1)**

**(= x3 x2)**

**)]**