Chapter 15-Algorithms

Monday, January 9, 2023
12:33 AM

Algorithms:

An Array provides constant-time access, or O(1), to any element.
However, to modify the array length or adding/deleting objects require *linear time*, or O(n).

In a ***Linked List***, each object points to the next object in the list. (elements can be accessed only by walking element-by-element from the head of the list)

The function *first* returns the first element of the list. E.g. first(L) returns 1.

The function *rest* returns the list missing its first element. E.g. rest(L) returns $(7, 3, 4, 7, 19)$.

The function *cons* adds a new element onto the head of the list, returning the new (longer) list. So cons(35,L) will return $(35, 1, 7, 3, 4, 7, 19)$.

Constant time to add, remove, or read/write the value **at the head** of the list, or to locations that are **constant distance from the head** (e.g. changing the 3rd element in the list, since no matter the size of the list the time it takes to reach the 3rd element is constant).
Linear time for adding, removing or accessing the values at other positions (e.g. changing the *n*th element, with n being a variable, is O(n)).

Nested Loops:
In simple programs that contain nested loops, the entire nested loop is **$O(n^2)$** because the inner loop executes *n* times and the outer loops executes *n* times.

A Connected Component Algorithm:
(definition in chapter 9)
The below pseudocode describes a program that returns all nodes in a connected component (of a specified graph) that has a specified node *s*

```
01  component(G: a graph; s: node in G)
02          Unmark all nodes of G.
03          RV = emptylist
04          Q = emptylist
05          Mark node s and add it to Q
06          while (Q is not empty)
07                  p = first(Q)
08                  Q = rest(Q)
09                  add p to RV
10                  for every node n that is a neighbor of p in G
11                          if n is not marked
12                                  mark n
13                                  add n to the tail of Q
14          return RV
```

(goes through all nodes and finds all of the neighbors of s, throws them into Q, then goes through each node in Q and finds all of their neighbors. Marking is to make sure you don't add the same node/nodes over and over)

Assume that the graph G has *n* nodes and *m* edges.
In the above program, the marking ensures that lines 7-9 run no more than *n* times; *lines 11-13 run once per edge traversed, but one edge might get traced twice, so these lines run no more than 2m times.*
In total, the program needs O(n + m) time.

*(Note that you **cannot** say the program is O(n) or O(m) since neither n nor m dominates the other. You could say it's O(n^2), but O(n + m) is probably more helpful)*

Binary Search:
A search algorithm that works by repeatedly dividing the search interval in half and only looking at the half that contains the desired answer (determined by comparing the midpoint).

Binary Search pseudocode that I just copy pasted:

```
01  squareroot(n: positive integer)
02          p = squarerootrec(n, 1, n)
03          if (n − p^2 ≤ (p + 1)^2 − n)
05                  return p
05          else return p + 1


11  squarerootrec(n, bottom, top: positive integers)
12          if (bottom = top) return bottom
13          middle = floor((bottom+top)/2)
14          if (middle^2 == n)
15                  return middle
16          else if (middle^2 ≤ n)
17                  return squarerootrec(n,middle,top)
18          else
19                  return squarerootrec(n,bottom,middle)
```

Although you might be too lazy to actually understand what the code is doing exactly, you can still tell that

The squareroot function takes only constant time since it only does simple arithmetic;

The squarerootrec function has 1 recursive call to itself, and each call only has a size half of the previous call;

And the non-recursive parts of squarerootrec takes only constant time.

So we can write the following definition for the run time of the binary search:

- $T(1) = c$
- $T(n) = T(n/2) + d$

(where c and d are constants representing the constant time operations, and T(n/2) represents the recursive call in squarerootrec)

Unroll the definition, and we get

$T(n) = T(\frac{n}{2^k}) + kd$

base case when $k = \log n$

So T(n) =

$c + d \log n$, which is $O(\log n)$

Merging Lists:

A recursive function that merges 2 **Sorted** Lists:

```
01  merge(L1,L2: sorted lists of real numbers)
02        if (L1 is empty)
03              return L2
04        else if (L2 is empty)
05              return L1
06        else if (head(L1) <= head(L2))
07              return cons(head(L1),merge(rest(L1),L2))
08        else
09              return cons(head(L2),merge(L1,rest(L2)))
```

We write the recursive definition:

- $T(1) = c$
- $T(n) = T(n-1) + d$

Unroll:

$nd + c$

Which is just O(n).

Merge Sort:

Divide an unmerged list by 2 until you have a bunch of tiny lists that only have 1 or 2 elements. Then the above merge algorithm becomes a sort algorithm (because 1-element lists are already "sorted", and 2-element lists can be sorted with the merge function in constant time).

Copy Pasta:

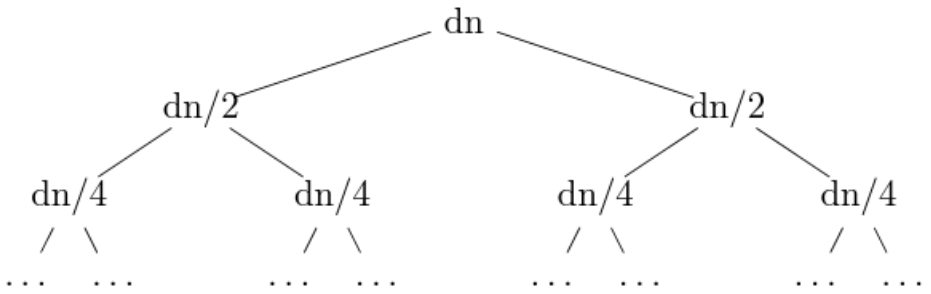
```
01  mergesort(L = a1, a2, ..., an: list of real numbers)
02        if (n = 1) then return L
03        else
04              m = ⌊n/2⌋
05              L1 = (a1, a2, ..., am)
06              L2 = (am+1, am+2, ..., an)
07              return merge(mergesort(L1),mergesort(L2))
```

There are 2 recursive calls to itself in mergesort, and in lines 5-6 O(n) time is needed to actually divide the list into 2. The merge function also takes O(n) time (explained above)

So we can write the following recursive definition for mergesort:

- $T(1) = c$
- $T(n) = 2T(n/2) + dn$

(where c is the base case constant, d is the number of times that the 2 O(n) parts of the function are used, and 2T(n/2) represents the 2 recursive calls of mergesort)
Tree:



The tree has $O(\log n)$ non-leaf levels and the work at each level sums up to dn. So the work from the non-leaf nodes sums up to $O(n \log n)$. In addition, there are n leaf nodes (aka base cases for the recursive function), each of which involves c work. So the total running time is $O(n \log n) + cn$ which is just $O(n \log n)$.

Note that when you are trying to come up with an algorithm and you want it to be efficient, you should try NOT to break up a big problem into 2 or more problems that are not much smaller than the original problem!
(e.g. breaking up a problem size n into 2 problems of size n-1)
(Although sometimes it is the only way to solve a specific problem)