Paul Kafka
9/26/12
COP 4530
Assignment 2

# Analysis

(i) The asymptotic time complexity for evaluation of the recursive function as a function of all arguments to the executable `recurse` which influence the running time of that program. For the analysis alone, assume that $S_1 = S_2 = 0$, $M_2 = M_4 = 1$, and $D_1 = D_2 > 1$. Briefly justify the results of your analysis.

$t(N) =$ A₁ + M₁*f(N/D₁) *Op* M₃*f(N/D₂) $=> t(N) = 1 + 2t\left(\frac{N}{D}\right)$

**Step 1**

$t(1) = 1$

$t(N) = 1 + 2t\left(\frac{N}{D}\right)$

$t\left(\frac{N}{D}\right) = 1 + 2t\left(\frac{N}{D^2}\right)$

$t\left(\frac{N}{D^2}\right) = 1 + 2t\left(\frac{N}{D^3}\right)$

$t\left(\frac{N}{D^3}\right) = 1 + 2t\left(\frac{N}{D^4}\right)$

**Step 2**

$t(N) = 1 + 2t\left(\frac{N}{D}\right)$

$= 1 + 2\left[1 + 2t\left(\frac{N}{d^2}\right)\right]$

$= 1 + 2 + 4t\left(\frac{N}{D^2}\right)$

$= 1 + 2 + 4\left[1 + 2t\left(\frac{N}{D^3}\right)\right]$

$= 1 + 2 + 4 + 8t(\frac{N}{D^3})$

**Step 3**

$\sum_{i=0}^{n} 2^i + 2^k\left(\frac{N}{D^k}\right)$

$= 2^k - 1 + 2^k\left(\frac{N}{D^k}\right)$

$= 2^k + 2^k t\left(\frac{n}{D^k}\right) - 1$

$= 2^k\left[1 + t\left(\frac{N}{D^k}\right) - 1\right]$

$= t(\frac{N}{D^k})$

**Step 4**

$N = D^k$

$\log_D N = k$

$2^{\log_D N} - 1 + 2^{\log_D N} * t\left(\frac{N}{D^{2^{\log_D N}}}\right)$

$D^{logD} = \frac{N}{N}$

$2^{\log_D N[1+1]-1}$

$2^{\log_D N+1}$

$O(2^{\log_D N})$

$O(2^{\log_2 N * \log_D 2})$

**Answer**

$O(N^{\log_D 2})$

(ii) Amortized time complexity for `N push` operations on the stack. Briefly justify your analysis.

| | |
|---|---|
| template<class T><br>void Stack< T >::push(T& p)<br>{Stacks.push_front(p);} | My Stack class used the push_front function from my vector class; this will put the added element at the top of the stack. The push_front function moves through the array in the loop n |

```cpp
template < class T >
void Vector< T >::push_front(const T &e)
// Adds and element to the front
{
    if(Size == Capacity)
        resize(4 * Capacity);       // resize if needed

    // shift the array
    T* temp = container;
    container = new T[Capacity];
    for(int i =0; i < Size; i++)
        container [i+1] = temp[i];
    delete [] temp;

    container[0] = e;   // insert element at the first slot
    Size++;             // increment the size
}
```

times at while pushing all the elements to the front n+1 times. This makes the amortized time complexity O(n). The resize function is negligible because it happens before the push_front does its job, and you don’t have to do it every time. The resize has a time complexity of O(1).