Theory Assignment

krbh@itu.dk, frai@itu.dk, jglr@itu.dk

March 2020

1

1.a $h_1(n) = n^{19}$

1.b $h_2(n) = n^{25}$

1.c $h_3(n) = n$

1.d True

2

Finding the limit of the ratio between the functions, we find that

$$\lim_{x \to \infty} \frac{n^{\log_2(n)}}{2^n} = 0$$

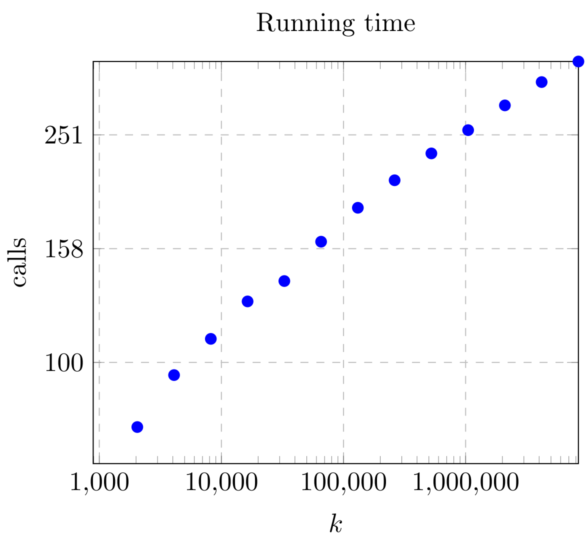
therefore 2^n grows faster than $n^{\log_2(n)}$
therefore $n^{\log_2(n)}$ is in $O(2^n)$

3

Found and modified algorithm by Vivek Kumar Singh

I was meant to argue for running time and correctness, but I can't due to time.

By my own empirical study, counting every time kth is called for doubling values of k, maybe suggests a running time of O(log k)



```
static int kth(int[] a, int aLen, int[] b, int bLen, int k) {
    // if k is out of range of the union between a and b returning -1
    if (k > (aLen + bLen) || k < 1)
      return -1;

    // set a to be the longest of the two arrays
    if (aLen > bLen)
      return kth(b, bLen, a, aLen, k);

    // if a (longest) is empty returning k-th largest element of b
    if (aLen == 0)
     return b[k - 1];

    // if k = 1 return maximum of first two elements of both arrays
    if (k == 1)
      return Math.max(a[0], b[0]);

    // divide and conquer
    int i = Math.min(aLen, k / 2);
    int j = Math.min(bLen, k / 2);

    if (a[i - 1] < b[j - 1]){
      // Now we need to find only k-j th element
      // since we have found out the lowest j
      int newB[] = Arrays.copyOfRange(b, j, bLen);
      return kth(a, aLen, newB, bLen - j, k - j);
    }

    // Now we need to find only k-i th element
    // since we have found out the lowest i
    int newA[] = Arrays.copyOfRange(a, i, aLen);
    return kth(newA, aLen - i, b, bLen, k - i);
  }
```

4

4.a

Sorts array A in ascending order, with insertion sort.

4.b

Worst case for this algorithm is an array sorted in descending order.

Line C_2 has one comparison and therefore costs 1.

Line C_3 has one swap and therefore costs 1.

All other lines has no cost. Every call of C_i costs 2i-2 in the worst case.

$$\begin{aligned} sum &= \frac{n(n+1)}{2} - 1 && (1) \\ &= \frac{n^2+n}{2} - 1 && (2) \end{aligned}$$

Average i notated $\bar{i}$

$$\bar{i} = \frac{\frac{n^2+n}{2} - 1}{n-1} \quad (3)$$

$$\begin{aligned} \sum_{i=2}^{n} 2i - 2 &= (2\bar{i} - 2) \cdot (n-1) && (4) \\ &= (2 \cdot \frac{\frac{n^2+n}{2} - 1}{n-1} - 2) \cdot (n-1) && (5) \\ &= (2(\frac{n}{2} + 1) - 2) \cdot (n-1) && (6) \\ &= (2 \cdot \frac{\frac{n^2+n}{2} - 1}{n-1} - 2) \cdot (n-1) && (7) \\ &= (2(\frac{n}{2} + 1) - 2) \cdot (n-1) && (8) \\ &= n \cdot (n-1) && (9) \\ &= n^2 - n && (10) \end{aligned}$$

The upper bound is therefore:

$$O(n) = n^2 - n$$

It's a tight upper bound because the worst case scenario has exactly the running time $n^2 - n$, therefore it's the smallest(tightest) upper bound possible.

5

5.a

A component with a single vertex has 0 edges.

For each added vertex to the component, while not violating the constraints of a tree, the component gets 1 more edge.

Therefore by induction, one tree has $v - 1$ edges.

And therefore a forest of k trees has $v - k$ edges.

5.b

If G had an odd-length cycle, implies that the start and end node of the cycle would both be adjacent and in the same partition. This would mean an edge from and to the same partition, violating the definition of bipartite.

6

6.a

3. is true

6.b

4. is true

6.c

The $m' = m + k$ approach works, but has different properties.

Increasing by $m' = m + k$ would mean that the fraction of free to occupied slots would asymptotically approach 0% as m tends to infinity.

Increasing by $m' = 2 \cdot m$ would mean that the fraction of free to occupied slots would remain somewhat constant. Somewhat constant depending on how often the hash table resizes.