CS 624: Analysis of Algorithms

Fall 2019 Assignment 1 – Solutions

1. As we saw in class, the generating function for the binomial series is: \((1 + x)^n = \sum_{k=0}^{n} \binom{n}{k}x^k\).

We can differentiate both sides with respect to \(x\) get \(n(1 + x)^{n-1} = \sum_{k=0}^{n} \binom{n}{k}kx^{k-1}\). Substitute \(x = 1\) and get \(n2^{n-1} = \sum_{k=1}^{n} \binom{n}{k}k\) (notice that the 0 term becomes 0)

2. a. \(n^2 = O(2^n)\) - True, since \(n^2 \leq c \times (2^n)\) for all \(n_0 \geq 2\) and \(c = 1\), for example.

   d. \(f(n) = O(g(n))\) implies \(2f(n) = O(2g(n))\) - False. See for example \(f=2n\) and \(g=n\).

3. This follows simply from the log change base rule. We know that \(\log_a x = \log_a b \times \log_b x\) for every two positive constants \(a, b\) and every \(x > 0\). Therefore \(\log_a b\) is our constant and any positive number can be our \(n_0\)

4. If \(f = O(g)\) then there are two constants \(c \) and \(n_0\) s.t. \(f(n) \leq c \times g(n)\) for all \(n \geq n_0\). If \(g = O(h)\) then there are two constants \(d\) and \(n_1\) s.t. \(g(n) \leq d \times h(n)\) for all \(n \geq n_1\). Combining the two, we get \(f(n) \leq c \times d \times h(n)\) for all \(n \geq \max(n_0, n_1)\), so we can use \(c \times d\) as our constant.

5. a. We can apply the master theorem here. We have \(n^{\log_2 2} = n\) and \(f(n) = n^4\). This is case 3 if we can show that there is a constant \(0 < c < 1\) and a constant \(n_0\) such that for every \(n > n_0\), \(a \times f(n/b) \leq c \times f(n)\), or in other words \(-2 \times n^2 \leq c \times n^4\). This is true for every \(n \geq 2\) and \(c = 1/2\), for example. Therefore \(T(n) = O(n^4)\).

   b. This falls into case 1 of the master theorem with \(a = 1\), \(b = 10/7\) and \(f(n) = n\). Since \(n^{\log_{10/7} 1} = 1 = \Omega(n)\), then \(T(n) = O(n)\).

   c. This is case 2 of the master theorem since \(n^{\log_4 16} = n^2\) and \(f(n) = n^2\), so the run time is \(O(n^2 \log n)\).

   f. We have \(a = 2, b = 4\) and \(f(n) = \sqrt{n}\). Since \(n^{\log_4 \sqrt{4}} = \Theta(\sqrt{n})\) this is case 2 and \(T(n) = O(\sqrt{n} \log n)\).

   g. We can show by induction that \(T(n) = O(n^3)\).

      - Base case: \(T(1) \leq cn^3\), OK if we consider a big enough \(c\).

      - Assume by induction that \(T(k) \leq ck^3\) for all \(1 \leq k < n\)

      - Substituting in the original equation we get: \(T(n) = c(n-2)^3 + n^2 = cn^3 - 6cn^2 + 12cn - 8c + n^2 = cn^3 - ((6c-1)n^2 - 12cn + 8c) \leq cn^3\) for any big enough \(c\), say \(c = 1\) and for any \(n \geq 2\), for example.

For a lower bound we can use the same trick in class, and as the inductive hypothesis use \(T(k) \leq c_1n^3 - c_2n^2\), and show that it works for the right choice of constants.
6. \[ T(n) = \sum_{j=2}^{n} (a + (j-1)c) = \sum_{j=2}^{n} a + c \sum_{j=2}^{n} (j-1) = a(n-1) + c \sum_{j=1}^{n-1} j = a(n-1) + c \frac{n(n-1)}{2}. \]

Rearranging the terms we get \( an - a + n^2 \frac{c}{2} - n \frac{c}{2} \). So, if we set \( A = \frac{c}{2}, B = a - \frac{c}{2} \) and \( C = -a \) we get a term of the form \( T(n) = An^2 + Bn + C \). Since \( c > 0 \), then \( A > 0 \) as well, so it’s a quadratic term.

7. See drawing. I can be quite flexible with any handling of terminal nodes where \( T(n/4) < 1 \).

8. Say we want to delete node \( i \). One can, for example, call \( \text{IncreaseKey}(A, i, \infty) \) followed by \( \text{ExtractMax}(A) \). The increase of the key to infinity will put it at the top of the heap, and the ExtractMax will take care of removing that key and arranging the heap. Both operations are \( O(\log n) \), so calling them one after the other is still logarithmic.

Another solution is to exchange the \( i^{th} \) number with the last, delete the last index by decreasing the heap size by 1. Then check if the new \( A[i] \) is smaller than its child(ren). If it is, call \( \text{Heapify}(A, i) \). If not – check whether it is not smaller than its parent. If so – float it up until it reaches its right place (as you would with increaseKey for example).

9. You can maintain a Min-heap \( A \) of size \( k \), that at every given moment keeps the minimum element in each of the \( k \) heaps. Each time we extract an element from the min-Heap we extract the minimum out of all these minima, and add it to the sorted list of merged items. Each time we put the minimum element in a heap we remove it from its list. We then have to trace back the list it came from and insert the next smallest element into the heap \( A \). We perform overall up to \( n \text{ HeapInsert} \) and \( n \text{ HeapExtractMin} \), but the size of the heap is always at most \( k \), so the total runtime is \( O(n \log k) \).