Data Structures and Algorithms Homework 2

Due Wednesday Sept 11; Joseph Sepich (jps6444)

1 Problem 1

1. $n^{\frac{1}{log(n)}}$
2. $log(log(n))$
3. $\sqrt{log(n)}$
4. $log^2(n)$
5. n^2
6. n^3
7. $n^{log(log(n))}$
8. $(\sqrt{2})^{log(n)}$
9. $2^{\sqrt{2log(n)}}$
10. $2^{log(n)}$
11. $(\frac{3}{2})^n$
12. $n2^n$
13. 2^{2^n}
14. $2^{2^{2n+1}}$
15. $log(n)!$
16. $log(n!)$
17. $n!$

Collaborators: None

2 Problem 2

2.1 Part a

How many sums and multiplications would it take to "brute force" calculating a polynomial?

We are working in big O notation, so we will consider the worst case scenario. In this scenario we will have n additions between the n+1 terms in the polynomial. Each of these n+1 terms will also have a multiplcation, except for the fist making it n multiplcations. However this number assumes we know x^k. Computing the exponential terms would require additional steps taking 0 then 1 then 2 then 3 all the way up to n multiplications. This would in fact create $\sum_{i=1}^{n} i + n$ total multiplcations. This gives us a total of $n + 1 + n + \frac{n(n+1)}{2}$ from Gauss's formula to sum consecutive whole numbers. This gives us a running time of $O(n^2)$.

Collaborators: None

2.2 Part b

Horner's rule works backwards. This algorithm works off the fact that with each new term, it has an additional value of x_0. Let's work from Horner's rule back to a polynomial.

1. The algorithm begins with $z = a_n$.
2. Next step: $z = a_n * x_0 + a_{n-1}$.
3. Next step $z = (a_n * x_0 + a_{n-1}) * x_0 = a_n * x_0^2 + a_{n-1} * x_0 $.
4. At this point you can see that x_0 is distributed over the rest of the calculated polynomial on each step. The loop goes from n-1 to 0, which totals to n iterations. With n iterations the term a_n would be multiplied by x_0 n times, which is the same as $a_n * x_0^n$.
5. Therefore Horner's rule must calculate the value of $p(x_0)$ if you continue the process described in step 4 for each term below it.

Collaborators: None

2.3 Part c

Horner's rule iterates from n-1 to 0 for a total of n iterations. There is exactly 1 summation and 1 multiplication in each iteration. This implies a running time of $n + n = 2n$ for a big O of O(n).

Collaborators: None

2.4 Part d

For the polynomial $p(x) = x^n$, Horner's rule will execute 2n additions/multiplications; however this particular polynomial is only a single term from a polynomial. Horner's will be adding 0 for each constant every time, and merely be multiplying x_0 to a beginning number of $a_n = 1$. A more efficient algorithm for this particular polynomial would be to calculate x^n using the divide and conquer approach we talked about in class. This method would only take O(nlog(n)) time.

Collaborators: None

3 Problem 3

Input:

- array A; size m; sorted numbers
- array B; size n; sorted numbers
- integer k

Ouput:

- k-th smallest element from A U B.

Want O(logn + logm) or O(log(nm)) running time. We know the naive brute force approach would be to merge like in merge sort, but this would only be O(n + m) time, which is not quick enough. The requested running time looks similar to a binary search, so let's perform a divide and conquer approach.

```
KSmall(int[] A, int[] B, int i, int j, int k) {
  // Start with base case
  // Assume i is index of A, and j is index of B
  // i + j = k-1 at zero based indexing

  int[] temp;

  lenA = len(A);
  lenB = len(B0);

  if(lenA + lenB < k) return -1; // kth element of union DNE

  iMin = 0;
  iMax = min(lenA, k-1);

  i = 0;
  j = 0;


  i = (iMin + iMax) / 2; // middle of array
  j = k - 1 - i; // from equation stated before

  if (B[j - 1] > A[i]) {
      return KSmall(A[i+1:k],B,k);
  } else if (i > 0 && A[i - 1] > B[j]) {
      return KSmall(A[0:i],B,k);
  } else {
      // i is perfect
     return Integer.min(A[i], B[j]);
  }

  return -1;
}
```

Collaborators: AlgorithmsAndMe.com

4 Problem 4

4.1 Part a

$T(n) = 11T(\frac{n}{5}) + 13n^{1.3}$

Let's use the master theorem.

$a = 11, b = 5, d = 1.3$ $log_b a = log_5 11 = 1.49 > d = 1.3$

Using case two we know $\Theta(n) = \Theta(n^{log_b a}) = \Theta(n^{1.49})$

Collaborators: None

4.2 Part b

$T(n) = 2T(\frac{n}{4}) + nlog(n)$

We know that in this algorithm we have a recurrence that divides the problem into 2 subproblems of size $\frac{n}{4}$. Each problem at that level has the work of nlog(n) implying the total work on each level is $4 * (\frac{n}{4}log(\frac{n}{4})) = nlog(\frac{n}{4})$. We also know that the total number of levels would be on the order of $log_4(n)$. Knowing the height of the tree and work on each level we can conclude a running time of $\Theta(nlog(n)log(\frac{n}{4}))$

Collaborators: None

4.3 Part c

$T(n) = 5T(\frac{n}{3} + log^2(n)$

We know that in this algorithm we have a recurrence that divides the problem into 5 subproblems of size $\frac{n}{3}$. Each problem at that level has the work of logˆ2(n) implying the total work on each level is $5 * (log^2(\frac{n}{3}))$. We also know that the total number of levels would be on the order of $log_3(n)$. Knowing the height of the tree and work on each level we can conclude a running time of $\Theta(log(n)log^2(\frac{n}{3}))$.

Collaborators: None

4.4 Part d

$T(n) = T(\frac{n}{2}) + 1.5^n$

We know that in this algorithm we have a recurrence that divides the problem into 1 subproblem of size $\frac{n}{2}$. Each problem at that level has the work of 1.5^n implying the total work on each level is $1.5^{\frac{n}{2}}$. We also know that the total number of levels would be on the order of $log(n)$. Knowing the height of the tree and work on each level we can conclude a running time of $\Theta(1.5^n log(n))$.

4.5 Part e

$T(n) = T(\sqrt{n}) + \Theta(loglog(n))$

We know that in this algorithm we have a recurrence that divides the problem into 1 subproblme of size $\sqrt{n}$. Each problem at that level ahs the work of $\Theta(loglog(n))$ implying the total work on each level is $\Theta(loglog(\sqrt{n}))$. We also know that the total number of levels would be on the order of loglog(n). Knowing the height of the tree and work on each level we can conclude a running time of $\Theta(loglog^2(\sqrt{n}))$.

Collaborators: None

4.6 Part f

$T(n) = 6T(n/2) + n^{2.8}$

Using Master Theorem we know $a = 6, b = 2, d = 2.8$. $log_b a = log_2 6 = 2.58 < d = 2.8$. Knowing this we can use case 1 and state the running time is $\Theta(n^{2.8})$

4.7 Part g

$T(n) = T(n/2 + \sqrt{n}) + 230$

Since the work at each step takes a constant amount of time, only the height of the recurrence tree matters. We know that it would be $\Theta(log(n))$ for T(n/2) and $\Theta(loglog(n))$ for $\sqrt{n}$, so we can then conclude the running time of this recurrnce would be $\Theta(log(n) + log(log(n)))$.

Collaborators: None

4.8 Part h

$T(n) = T(n-2) + log(n)$

The work at each step is log(n), but the subproblem of each leaf is n-2. The size of this subproblem means the heigh of this tree would be n/2. From this we can conclude a running time of $\Theta(nlog(n))$.

Collaborators: None

4.9 Part i

$T(n) = T(3n/5) + T(2n/5) + \Theta(n)$

From this recurrence running time we know that each step takes $\Theta(n)$ time to complete. We also know that there are 2 subproblems created at each step, one of size 3n/5 and one of size 2n/5. This would imply a tree height of log(n), which allows us to conclude a running time of $\Theta(nlog(n))$

Collaborators: None

4.10 Part j

$T(n) = \sqrt{n}T(\sqrt{n}) + 30n$

From this recurrence running time we know that each step has 30n steps of work. We also know that there are $\sqrt{n}$ subproblems of $\sqrt(n)$ in size. This implies a height of log(log(n)). From here we can conclude a running time of $\Theta(nlog(log(n)))$

Collaborators: None

5 Problem 5

Input:

- matrix H_k; size 2^k by 2^k
- vector v; size 2^k

Output:

- vector b

Let's recall the standard method of calculating vector-matrix multiplcation. Using i as row and j as column index, $b(i) = \sum_{z=1}^{2^k} \sum_{x=1}^{2^k} H_k(z,x) * v(x)$. Using this "brute force" algorithm it would take you n by n multiplications and slightly less summations to calculate the answer. This would be $O(n^2)$, but we can do better. As stated we are working with Hadamard matrices, not an unknow matrix. These matrices following the recursive pattern detail above.

Since the matrix H_k is comprised of 4 submatrices of H_{k-1}, we can divide the problme into 4 smaller subproblems. We can actually take this further and only use 2 subproblems, because 3 of the matrices are identical, and one is the negative value of the others. Knowing that the size of the matrix is $n = 2^k$, we know the subproblem size is n/2. ($2^{k-1} = \frac{2^k}{2} = \frac{n}{2}$). The only thing we don't know yet is the work of at each level, but we know that the work comes with combinging the answers, since we can reduce down to a 1x1 matrix to perform a single multiplcation. Combining the answers would then take linear time. Therefore our running time recursively can be described as $T(n) = 2T(\frac{n}{2}) + O(n)$. Solving the running time with the Master theorem, we have $a = 2, b = 2, d = 1$. $log_b a = log_2 2 = 1 = d = 1$. We can then use case 2: $\Theta(nlog(n))$

Collaborators: None