Homework 3

Friday, March 12, 2021 6:52 PM

1. Recurrence [30 pt]

a) Suppose you have to choose among the follow three algorithms:

- Algorithm A solves the problem by dividing it into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm B solves the problem of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.
- Algorithm C solves problems of size n by dividing them into seven problems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $\boldsymbol{O}(n^2)$ time.

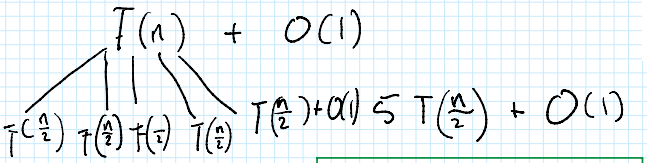
What are the running times of each of these algorithms (in big-$\boldsymbol{O}$ notation). Which algorithm would you choose? **[10 pt]**

Masters theorem

$T(n) = aT(n/b) + O(n^d)$ for $a \geq 1$, integer $b > 1$, and $d \geq 0$.

1. $a < b^d$: $T(n) = O(n^d)$
2. $a = b^d$: $T(n) = O(n^d \log n)$
3. $a > b^d$: $T(n) = O(n^{\log_b a})$

Algorithm A: $T(n) = 5T(\frac{n}{2}) + O(1)$



using masters theorem

$a = 5$, $b = 2$, $d = 0$

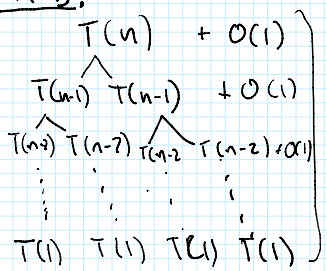
$a > b^d$

$5 > 2^0$

$5 > 1$ case #3

Algorithm A: $T(n) = O(n^{\log_2 5})$

Algorithm B:



$$T(n) = \begin{cases} 1 & n = 0 \\ 2T(n-1) + O(1) & n > 0 \end{cases}$$

Substitution method

$T(n) = 2T(n-1) + O(1) = 2^1 T(n-1) + O(1)$

$2T(2T(n-2)) + O(1) = 2^2 T(n-2) + O(1)$

$2T(2T(2T(n-3))) + O(1) = 2^3 T(n-3) + O(1)$

$\vdots$

$2^k T(n-k) + O(1)$

Geometric relationship $2^k = \frac{2^{k+1}-1}{2-1} = 2^{k+1} - 1$

Assume $n - k = 0$ is base case, > 0

$n = k$

Substituting n for k

$2^{n+1} - 1$ which is in $O(2^n)$

Algorithm B: $O(2^n)$

Algorithm C: $T(n) = 7T(\frac{n}{3}) + O(n^2)$

using masters theorem $a = 7$, $b = 3$, $d = 2$

$a < b^d$ $\quad 6^d = 3^2 = 9$

$7 < 9$ case #1

$O(n^d) = O(n^2)$ ← this makes sense because $O(n^2)$ would eliminate all other terms.

Based on the above, you should pick Algorithm C because it runs in $O(n^2) <$ A which is in $\simeq O(n^{2.3219}) <$ B which is in $O(2^n)$

b) Give the *tightest asymptotic upper bound* (in $O(\cdot)$) you can obtain for the following recurrences. Justify your answer. **[5 pt each]**

- $T(n) = 2T(n/4) + 330\sqrt{n}$

Using Masters theorem if $a = 2$, $b = 4$, $d = 1/2$

$330\sqrt{n} = O(\sqrt{n}) = O(n^{1/2})$

$b^d = 4^{1/2} = 2$

$a = b^d$

$2 = 4^{1/2}$

$2 = 2$

case #2 $O(n^d \log n)$

$O(\sqrt{n} \log n)$

- $T(n) = 20T(n/15) + n \log n$

$T(n) = \begin{cases} 1 & n = 1 \\ 20T(n/15) + n \log n \end{cases}$

$a = 20$, $b = 15$, $k = 1$

$\log_{15} 20 \approx 1.11$

$1.11 \approx \log_{15} 20 > k = 1$

$\Theta = (n^{\log_{15} 20})$

Using ↓

General Master Theorem for Dividing Functions

$T(n) = aT(n/b) + f(n)$ $a \geq 1$ $b > 1$ $f(n) = O(n^k \log^p n)$

① $\log_b a$

② k

case 1: if $\log_b a > k$ then $\Theta(n^{\log_b a})$

case 2: if $\log_b a = k$

if $p > -1$ $O(n^k \log^{p+1} n)$

$p = -1$ $O(n^k \log \log n)$

$p < -1$ $O(n^k)$

case 3: if $\log_b a < k$

if $p \geq 0$ $O(n^k \log^p n)$

$p < 0$ $O(n^k)$

- $T(n) = T(n-1) + 3 \log n$, $T(1) = 1$

$a = 1$, $k = 0$ case 2 $O(n^{k+1})$ ⇒ using this

$T(n) = \Theta(n)$ ← because $O(n) > O(\log n)$

General Master Theorem for Decreasing functions

$T(n) = aT(n-b) + f(n)$

$a > 0$ $b > 0$ and $f(n) = O(n^k)$ where $k \geq 0$

case #1 $a < 1$ $O(n^k)$ or $O(f(n))$

case #2 if $a = 1$ $O(n^{k+1})$

case #3 if $a > 1$ $O(n^k a^n)$

$O(f(n) a^{n/b})$

- $T(n) = (\log n) T(n/2) + 1$

using general master theorem for dividing functions

$a = \log n$

$b = 2$

$k = 0$

$f(n) = 1 = O(n^0) = O(1)$

$\log \log n > 0$ so case #1 $O(n^{\log_b a})$

and $\log_b a = \log \log n$

so

$T(n) = O(n^{\log \log n})$

2. Sort [20 pt]

a) Write down the recurrence relation of the following sorting algorithm. What is the time complexity of this method (in $O(\cdot)$ notation)? Justify your answer. **[10 pt]**

SS (n min max)

n

```
void StrangeSort(int a[], int min, int max) {
    if (min >= max)          ← exit loop/base case
        return;
    if (a[min] > a[max])
        swap(a[min], a[max]);  // constant-time operation
    int one_third = (max - min + 1) / 3;
    if (one_third >= 1) {
        StrangeSort(a[], min, max - one_third);
        StrangeSort(a[], min + one_third, max);
        StrangeSort(a[], min, max - one_third);
    }
}
```

1 ← if (a[min] > a[max])

1 ← int one_third = (max - min + 1) / 3;

SS (2/3 n) ← StrangeSort(a[], min, max - one_third);

SS (2/3 n) ← StrangeSort(a[], min + one_third, max);

SS (2/3 n) ← StrangeSort(a[], min, max - one_third);

SS (2/3 n)

SS(2/3 n) ← StrangeSort(a[], min, max - one_third);
}
}

a[8, 6, 7, 5, 3, 4] min=0 max=5

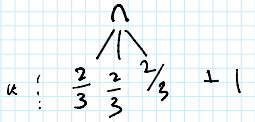
if a[0] > a[5]
8 4
swap

a[4, 6, 7, 5, 3, 8]

one_third = (5 - 0 + 1)/3 = 6/3 = 2

one_third >= 1

SS(a[], 0, 4]
SS(a[], 2, 5]
SS(a[], 0, 4]



$T(n) = 3T(\frac{2}{3}n) + 1$

using general master theorem

$a = 3$ $f_n = n^0 = 1$

$b = 3/2$ $k = 0$

$\log_b a = \log_{3/2} 3 \simeq 2.7095$

$2.7095 > 0$

$\boxed{O(n^{\log_{3/2} 3})}$ case #1 Gen. Master Theorem

b) Write down the recurrence relation of the following sorting algorithm. What would be the best-case input for this algorithm? For that case, is it faster or slower than Bubble Sort? **[10 pt]**

```
void MysterySort(int a[], int min, int max) {
      if (min >= max)   ← exit loop/base case
            return;
      int mid = floor((min + max)/2);
T(n/2) ←MysterySort(a[], min, mid);
T(n/2) ←MysterySort(a[], mid+1, max);
      ← if (a[max] < a[mid])
            swap(a[max], a[mid])  // constant-time operation
T(n-1) ←MysterySort(a[], min, max-1);
}
```

Trace of Program → MS(a[0, 8, 2, 1, 7], min=0, max=4)

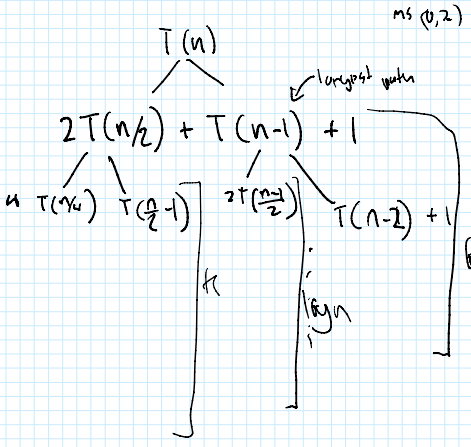
mid = (0+5/2) = 2

MS(a[], 0, 2] → MS(0,1), MS(2,2)

MS[a[], 2, 4]

if(a[0] < a[4])
0 7
no swap

MS[a[], 0, 3] → MS(0,1), MS(1,3), MS(0,2)



using general master theorems if $2T(n/2) + f(n)$ and if $f(n) = T(n-1) + g(n)$

then $\log_b a = \log_2 2 = 1$

& $g(n) = 2(\frac{n-1}{2})$

$k=1$ then $f(n) = O(n^1)$

$= 4T(\frac{n}{4}) + T(\frac{n}{2} - 1) + 2T(\frac{n-1}{2}) + T(n-2) + 2$ so $\log_b a = k$

$O(n)$

$\boxed{T(n) = O(n \log^2 n)}$

So case 2 where $p > -1$

$O(n^k \log^{p+1} n)$

$O(n \log^2 n)$

There isn't a best case for this since it will sort this way regardless of the input but for a large unsorted n this is better than bubble sort. As the array is more sorted bubble sort is better