COMP20007 Design of Algorithms

Week 8 Workshop Solutions

Tutorial

1. Master Theorem Note for this question that comparing a and b^d is the same as comparing $\log_b a$ and d.

(a) $T(n) = 9T\left(\frac{n}{3}\right) + n^3,\ T(1) = 1$

We have $a = 9, b = 3, d = 3$ and $c = 1$. So $\log_b(a) = \log_3(9) = 2$.

Also $2 < 3 = c$, so the $\Theta(n^3)$ is the dominating term. So $T(n) \in \Theta(n^3)$.

(b) $T(n) = 64T\left(\frac{n}{4}\right) + n + \log n,\ T(1) = 1$

We have $a = 64$ and $b = 4$, so $\log_b(a) = \log_4(64) = 3$.

Also, since $n + \log n \in \Theta(n^1)$, $d = 1 < 3$, so $T(n) \in \Theta(n^3)$.

(c) $T(n) = 2T\left(\frac{n}{2}\right) + n,\ T(1) = 1$

We have $a = b = 2$ so $\log_b(a) = \log_2(2) = 1$. Also $n \in \Theta(n^1)$ so $d = 1$ as well.

So $T(n) \in \Theta(n^d \log n) = \Theta(n \log n)$.

✓(d) $T(n) = 2T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2}\right) + \Theta(1),\ T(1) = 1$

Here $a = b = 2$ and $d = 0$ so $b^d = 1 < 2 = a$, thus we get $\Theta(n^{\log_2 2}) = \Theta(n)$.

2. Simple Sorting Algorithms

(a) *Selection Sort.* When running selection sort the key idea is that for each i from 0 to $n-2$ we find the smallest (or largest if we're sorting in descending order) element in $A[i \dots n-1]$ and swap it with $A[i]$.

(i) We'll show the elements at the start of the array which we have already sorted (i.e., the elements $A[0 \ldots i-1]$) in bold, and the minimum element in $A[i \ldots n-1]$ in red.

$$\begin{bmatrix} \textcolor{red}{\mathrm{A}} & \mathrm{N} & \mathrm{A} & \mathrm{L} & \mathrm{Y} & \mathrm{S} & \mathrm{I} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathrm{N} & \mathrm{A} & \mathrm{L} & \mathrm{Y} & \mathrm{S} & \mathrm{I} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathrm{N} & \textcolor{red}{\mathrm{A}} & \mathrm{L} & \mathrm{Y} & \mathrm{S} & \mathrm{I} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathrm{N} & \mathrm{L} & \mathrm{Y} & \mathrm{S} & \mathrm{I} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathrm{N} & \mathrm{L} & \mathrm{Y} & \mathrm{S} & \textcolor{red}{\mathrm{I}} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathbf{I} & \mathrm{L} & \mathrm{Y} & \mathrm{S} & \mathrm{N} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathbf{I} & \textcolor{red}{\mathrm{L}} & \mathrm{Y} & \mathrm{S} & \mathrm{N} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathbf{I} & \mathbf{L} & \mathrm{Y} & \mathrm{S} & \mathrm{N} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathbf{I} & \mathbf{L} & \mathrm{Y} & \mathrm{S} & \textcolor{red}{\mathrm{N}} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathbf{I} & \mathbf{L} & \mathbf{N} & \mathrm{S} & \mathrm{Y} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathbf{I} & \mathbf{L} & \mathbf{N} & \textcolor{red}{\mathrm{S}} & \mathrm{Y} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathbf{I} & \mathbf{L} & \mathbf{N} & \mathbf{S} & \mathrm{Y} & \mathrm{S} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathbf{I} & \mathbf{L} & \mathbf{N} & \mathbf{S} & \mathrm{Y} & \textcolor{red}{\mathrm{S}} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{A} & \mathbf{A} & \mathbf{I} & \mathbf{L} & \mathbf{N} & \mathbf{S} & \mathbf{S} & \mathrm{Y} \end{bmatrix}$$

(ii) At each $i \in \{0, \ldots, n-2\}$ we must take the minimum element from the array $A[i \ldots n-1]$, which requires $n-1-i$ comparisons. So the total number of comparisons, $C(n)$, will be:

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2}(n-1-i) \\ &= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i \\ &= (n-1)(n-1) - \frac{(n-2)(n-1)}{2} \\ &= \frac{1}{2}\left(2n^2 - 4n + 2 - n^2 + 3n - 2\right) \\ &= \frac{1}{2}\left(n^2 - n\right) \\ &\in \Theta(n^2) \end{aligned}$$

(iii) Selection sort is **not** stable. Consider the following counter example (where 1_a and 1_b are used to differentiate between the two elements with the same value).

$$\begin{bmatrix} 1_a & 1_b & 0 \end{bmatrix}$$
$$\begin{bmatrix} 1_a & 1_b & \color{red}{0} \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{0} & 1_b & 1_a \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{0} & \color{red}{1_b} & 1_a \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{0} & \mathbf{1_b} & 1_a \end{bmatrix}$$

Since 1_a and 1_b end up out of order relative to one another, this sorting algorithm is *not* stable.

(iv) Yes, selection sort sorts *in-place*, as it requires only $O(1)$ additional memory.

(v) No selection sort is not input sensitive, the number of comparisons is a function of n only and the order of the elements has no effect.

(b) *Insertion Sort.* The key idea of insertion sort is that we have some section at the start of the array which is sorted (initially only $A[0]$) and we repeatedly *insert* the next element into the correct position in this sorted segment, until the whole array is sorted.

More precisely, with i going from 1 to $n-1$ we have $A[0 \dots i-1]$ already sorted, and we swap $A[i]$ backwards until it is in the correct position in the sorted section $A[0 \dots i]$ (*i.e.*, it's greater then the element immediately preceding it).

(i) For each i we'll show the sorted section in bold, and then when we're performing the insertion we'll show the element we're inserting in red, and the element we're comparing/swapping

$$\left[\mathbf{A}\ \mathrm{N\ A\ L\ Y\ S\ I\ S}\right]$$
$$\left[\mathrm{A\ N\ A\ L\ Y\ S\ I\ S}\right]$$
$$\left[\mathbf{A\ N}\ \mathrm{A\ L\ Y\ S\ I\ S}\right]$$
$$\left[\mathrm{A\ N\ A\ L\ Y\ S\ I\ S}\right]$$
$$\left[\mathrm{A\ A\ N\ L\ Y\ S\ I\ S}\right]$$
$$\left[\mathrm{A\ A\ N\ L\ Y\ S\ I\ S}\right]$$
$$\left[\mathbf{A\ A\ N}\ \mathrm{L\ Y\ S\ I\ S}\right]$$
$$\left[\mathrm{A\ A\ N\ L\ Y\ S\ I\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ N\ Y\ S\ I\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ N\ Y\ S\ I\ S}\right]$$
$$\left[\mathbf{A\ A\ L\ N}\ \mathrm{Y\ S\ I\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ N\ Y\ S\ I\ S}\right]$$
$$\left[\mathbf{A\ A\ L\ N\ Y}\ \mathrm{S\ I\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ N\ Y\ S\ I\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ N\ S\ Y\ I\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ N\ S\ Y\ I\ S}\right]$$
$$\left[\mathbf{A\ A\ L\ N\ S\ Y}\ \mathrm{I\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ N\ S\ Y\ I\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ N\ S\ I\ Y\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ N\ S\ I\ Y\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ N\ I\ S\ Y\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ N\ I\ S\ Y\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ I\ N\ S\ Y\ S}\right]$$
$$\left[\mathrm{A\ A\ L\ I\ N\ S\ Y\ S}\right]$$
$$\left[\mathrm{A\ A\ I\ L\ N\ S\ Y\ S}\right]$$
$$\left[\mathbf{A\ A\ I\ L\ N\ S\ Y}\ \mathrm{S}\right]$$
$$\left[\mathrm{A\ A\ I\ L\ N\ S\ Y\ S}\right]$$
$$\left[\mathrm{A\ A\ I\ L\ N\ S\ S\ Y}\right]$$
$$\left[\mathbf{A\ A\ I\ L\ N\ S\ S\ Y}\right]$$

(ii) The best case for insertion sort is when the array is already sorted and there are no swaps to be made. There are exactly $n-1$ comparisons made in this case, thus insertion sort is $\Omega(n)$.

On the other hard, the worst case input is when the input is sorted in reverse order, and for each $i \in \{1, \ldots, n-1\}$ there are i swaps to be made (and hence i comparisons), thus the number of comparisons is:

$$C(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2).$$

So the worst case time complexity of insertion sort is $O(n^2)$.

(iii) Insertion sort *is stable.* The only way two elements relative order will be reversed is if they are swapped with each other directly. Since we do not swap elements of equal value it must be the case that relative orderings of elements with equal value must be maintained.

(iv) Insertion sort does sort in place, there is only a constant amount of additional memory required.

(v) Insertion sort is input sensitive, as we can see from (iii) where we show that the time complexity is different depending on how the input is arranged.

(c) *Quicksort (with Lomuto partitioning).* Quicksort is a recursive algorithm which repeatedly partitions the input array and recursively quicksorts the left and right portions of the array which $< p$ and $\geq p$ for the pivot p, respectively.

The *Quicksort* algorithm is:

```
function QUICKSORT($A[l \ldots r]$)
    if $l < r$ then
        $s \leftarrow$ PARTITION($A[l \ldots r]$)
        QUICKSORT($A[l \ldots s-1]$)
        QUICKSORT($A[s+1 \ldots r]$)
```

The *Lomuto* partitioning algorithm is:

```
function LOMUTOPARTITION($A[l \ldots r]$)
    $p \leftarrow A[l]$
    $s \leftarrow l$
    for $i \leftarrow l+1$ to $r$ do
        if $A[i] < p$ then
            $s \leftarrow s+1$
            SWAP($A[s], A[i]$)
    SWAP($A[l], A[s]$)
    return $s$
```

(i) Performing the algorithm on [A N A L Y S I S]:

Partition [A N A L Y S I S]

$$\left[\mathrm{A}_s\ \mathrm{N}_i\ \mathrm{A\ L\ Y\ S\ I\ S}\right] \to \left[\mathrm{A}_s\ \mathrm{N}\ \mathrm{A}_i\ \mathrm{L\ Y\ S\ I\ S}\right] \to \ldots \to \left[\mathrm{A}_s\ \mathrm{N\ A\ L\ Y\ S\ I}\ \mathrm{S}_i\right]$$

So $s = 1$ and we're left with (where bold characters are already fixed in place):

$$\left[\mathbf{A}\ \mathrm{N\ A\ L\ Y\ S\ I\ S}\right]$$

So partitioning [N A L Y S I S]:

$$\begin{gathered}\Big[\mathrm{N}_s\ \mathrm{A}_i\ \mathrm{L\ Y\ S\ I\ S}\Big] \to \Big[\mathrm{N}\ \mathrm{A}_{s,i}\ \mathrm{L\ Y\ S\ I\ S}\Big] \to \Big[\mathrm{N}\ \mathrm{A}_s\ \mathrm{L}_i\ \mathrm{Y\ S\ I\ S}\Big] \\ \to \Big[\mathrm{N\ A}\ \mathrm{L}_{s,i}\ \mathrm{Y\ S\ I\ S}\Big] \to \Big[\mathrm{N\ A}\ \mathrm{L}_s\ \mathrm{Y}_i\ \mathrm{S\ I\ S}\Big] \to \Big[\mathrm{N\ A}\ \mathrm{L}_s\ \mathrm{Y}\ \mathrm{S}_i\ \mathrm{I\ S}\Big] \\ \to \Big[\mathrm{N\ A}\ \mathrm{L}_s\ \mathrm{Y\ S}\ \mathrm{I}_i\ \mathrm{S}\Big] \to \Big[\mathrm{N\ A\ L}\ \mathrm{Y}_s\ \mathrm{S}\ \mathrm{I}_i\ \mathrm{S}\Big] \\ \to \Big[\mathrm{N\ A\ L}\ \mathrm{I}_s\ \mathrm{S}\ \mathrm{Y}_i\ \mathrm{S}\Big] \to \Big[\mathrm{N\ A\ L}\ \mathrm{I}_s\ \mathrm{S\ Y}\ \mathrm{S}_i\Big] \to \Big[\mathrm{N\ A\ L}\ \mathrm{I}_s\ \mathrm{S\ Y\ S}\Big] \to \Big[\mathrm{I\ A\ L}\ \mathrm{N}_s\ \mathrm{S\ Y\ S}\Big]\end{gathered}$$

So we have,

$$\Big[\mathbf{A}\ \mathrm{I\ A\ L}\ \mathbf{N}\ \mathrm{S\ Y\ S}\Big]$$

So we need to recursively Quicksort [I A L] and [S Y S].

Starting with [I A L]:

$$\begin{gathered}\Big[\mathrm{I}_s\ \mathrm{A}_i\ \mathrm{L}\Big] \to \Big[\mathrm{I}\ \mathrm{A}_{s,i}\ \mathrm{L}\Big] \to \Big[\mathrm{I}\ \mathrm{A}_s\ \mathrm{L}_i\Big] \\ \to \Big[\mathrm{I}\ \mathrm{A}_s\ \mathrm{L}\Big] \to \Big[\mathrm{A}\ \mathrm{I}_s\ \mathrm{L}\Big]\end{gathered}$$

So we have [A I L], and we recursively quicksort [A] and [L] (the base case, so we do nothing).

Now we have [**A A I L N** S Y S] and we need to quicksort [S Y S]:

$$\Big[\mathrm{S}_s\ \mathrm{Y}_i\ \mathrm{S}\Big] \to \Big[\mathrm{S}_s\ \mathrm{Y}\ \mathrm{S}_i\Big] \to \Big[\mathrm{S}_s\ \mathrm{Y\ S}\Big]$$

Nothing was moved, but we now have [**A A I L N S** Y S] and we need to recursively quicksort [Y S]:

$$\Big[\mathrm{Y}_s\ \mathrm{S}_i\Big] \to \Big[\mathrm{Y}\ \mathrm{S}_{s,i}\Big] \to \Big[\mathrm{Y}\ \mathrm{S}_s\Big] \to \Big[\mathrm{S}\ \mathrm{Y}_s\Big]$$

So we have [**A A I L N S** S **Y**] and the final recursive call for [Y] hits the base case, and we're finished! So the final sorted array is [A A I L N S S Y]

(ii) From the lectures Quicksort is $\Theta(n \log n)$ in the best case and $\Theta(n^2)$ in the worst case (for example when the array is in reverse sorted order we do n partitions of cost $\Theta(n)$).

(iii) Quicksort with Lomuto partitioning is not stable. Consider the counter example:

$$\Big[2\ 1^a\ 1^b\Big] \to \Big[2_s\ 1^a_i\ 1^b\Big] \to \Big[2\ 1^a_{s,i}\ 1^b\Big] \to \Big[2\ 1^a_s\ 1^b_i\Big] \to \Big[2\ 1^a\ 1^b_{s,i}\Big] \to \Big[2\ 1^a\ 1^b_s\Big] \to \Big[1^b\ 1^a\ 2_s\Big]$$

After the first partition we're left with [1^b 1^a] to recursively quicksort. Partitioning this with Lomuto partitioning leaves it as is (check this!) and so our input [2 1^a 1^b] becomes [1^b 1^a 2] after being quicksorted. The relative order of the 1s was not preserved so this algorithm must not be stable.

(iv) The algorithm is in place. Here we're allowing the $O(\log n)$ space required for the function stack when we do the recursive calls.

(v) This algorithm is input sensitive as demonstrated by the different best and worst case time complexities depending on the input order.

3. Mergesort Time Complexity Recurrence relation:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

where $T(n)$ is the runtime of mergesort sorting n elements. The first $T(\frac{n}{2})$ is the time it takes to sort the left half of the input using mergesort. The other $T(\frac{n}{2})$ is the time it takes to sort the right half. $\Theta(n)$ is a bound on the time it takes to merge the two halves together.

Recall that the Master Theorem states that if we have a recurrence relation $T(n)$ such that

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + \Theta(n^d), \\ T(1) &= c, \end{aligned}$$

then,

$$T(n) \in \begin{cases} \Theta\left(n^d\right) & \text{if } a < b^d \\ \Theta\left(n^d \log n\right) & \text{if } a = b^d \\ \Theta\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}.$$

We can recognise that the mergesort recurrence relation fits the form required by the Master Theorem, with constants $a = 2$, $b = 2$, and $d = 1$.

$$b^d = 2 = a$$

so, by the master theorem, $T(n) \in \Theta(n \log n)$.

4. Lower bound for the Closest Pairs problem We can use the closest pair algorithm from class to solve the element distinction problem like so, where the output is a collection of elements $C = \{c_1, \ldots, c_n\}$ and the output is Distinct or NotDistinct:

```
function ElementDistinction($C = \{c_1, \ldots, c_n\}$)
    $Points \leftarrow \{(c_1, 0), \ldots, (c_n, 0)\}$
    $Distance \leftarrow$ ClosestPair($Points$)
    if $Distnace$ is 0 then
        return NotDistinct
    else
        return Distinct
```

So, we can see that we can solve the element distinct problem using the closest pair algorithm. This is called a *reduction* from element distinction to closest pair.

We know that ElementDistinction is $\Omega(n \log n)$, and want to prove that ClosestPair is also $\Omega(n \log n)$.

We assume for the sake of contradiction that ClosestPair can be solved in a time complexity smaller (*i.e.*, asymptotically faster) that $n \log n$. As we have exhibited (provided) a reduction from Element-Distinction to ClosestPair then this must also give us an algorithm for ElementDistinction which is asymptotically faster than $n \log n$.

This contradicts the statement that ElementDistinction is $\Omega(n \log n)$, and as a result our assumption that ClosestPair can be solved in a time complexity smaller (*i.e.*, asymptotically faster) that $n \log n$ must be false.

Hence ClosestPair can not be solved in faster than $n \log n$ time, and is therefore $\Omega(n \log n)$.