COMP20007 Design of Algorithms

Week 9 Workshop Solutions

Tutorial

1. 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 $A[0 \dots i-1]$) in bold, and the minimum element in $A[i \dots n-1]$ in red.

$$\begin{bmatrix} \text{A N A L Y S I S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A} \text{ N A L Y S I S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A} \text{ N A L Y S I S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A A} \text{ N L Y S I S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A A} \text{ N L Y S I S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A A I} \text{ L Y S N S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A A I} \text{ L Y S N S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A A I L} \text{ Y S N S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A A I L} \text{ Y S N S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A A I L N} \text{ S Y S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A A I L N} \text{ S Y S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A A I L N S} \text{ Y S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A A I L N S} \text{ Y S} \end{bmatrix}$$
$$\begin{bmatrix} \textbf{A A I L N S S} \text{ Y} \end{bmatrix}$$

(ii) At each $i \in \{0, \dots, n-2\}$ we must take the minimum element from the array $A[i \dots n-1]$,

which requires $n-1-i$ comparisons. So the total number of comparisons, $C(n)$, will be:

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

(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).

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

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 $A[0 \ldots i-1]$ already sorted, and we swap $A[i]$ backwards until it is in the correct position in the sorted section $A[0 \ldots i]$ (*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 compar-

ing/swapping

[**A** N A L Y S I S]

[A N A L Y S I S]

[**A N** A L Y S I S]

[A N A L Y S I S]

[A A N L Y S I S]

[A A N L Y S I S]

[**A A N** L Y S I S]

[A A N L Y S I S]

[A A L N Y S I S]

[A A L N Y S I S]

[**A A L N** Y S I S]

[A A L N Y S I S]

[**A A L N Y** S I S]

[A A L N Y S I S]

[A A L N S Y I S]

[A A L N S Y I S]

[**A A L N S Y** I S]

[A A L N S Y I S]

[A A L N S I Y S]

[A A L N S I Y S]

[A A L N I S Y S]

[A A L N I S Y S]

[A A L I N S Y S]

[A A L I N S Y S]

[A A I L N S Y S]

[**A A I L N S Y** S]

[A A I L N S Y S]

[A A I L N S S Y]

[**A A I L N S S Y**]

(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 ... r])
    if l < r then
        s ← PARTITION(A[l ... r])
        QUICKSORT(A[l ... s − 1])
        QUICKSORT(A[s + 1 ... r])
```

The *Lomuto* partitioning algorithm is:

```
function LOMUTOPARTITION(A[l ... r])
    p ← A[l]
    s ← l
    for i ← l + 1 to r do
        if A[i] < p then
            s ← 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]

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

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

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

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

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

So we have,

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

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

Starting with [I A L]:

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

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]:

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

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.

2. Longest Common Substring Problem

(i) One exhaustive approach is to iterate through one of the strings (let's call this s_1), starting from each character in s_1, we iterate through starting at every letter in s_2, and count how many characters in s_1 match from the start of the string, keeping track of the longest match.

(ii) The worst case complexity of this exhaustive algorithm is $\Theta(m^2 n)$ where n is the length of the longer string and m is the length of the shorter string. Though this worst case only occurs when the pattern and string being searched are the same.

(iii) The subproblem which can be solved is that for the kth character, the preceding $k-1$ characters are checked multiple times.

(iv) If the length of s_1 is n and the length of s_2 is m, we can build an $n \times m$ table where each element is given by:

$$T_{i,j} := \begin{cases} 0 & \text{if } s_1[i] \neq s_2[j], \\ 1 & \text{if } s_1[i] = s_2[j] \text{ and } (i = 0 \text{ or } j = 0), \\ T_{(i-1),(j-1)} + 1 & \text{otherwise.} \end{cases}$$

For the given problem when run this gives:

$$T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 \end{bmatrix}$$

With the largest value occurring in the matrix occurring at $T_{5,8} = 4$

(v) The complexity is simply a loop over all i, setting the value for each j, hence the complexity is $\Theta(nm)$.

3. (Optional) Quickselect *Quickselect* is an algorithm for solving the kth-smallest element problem using the PARTITION algorithm from quicksort.

(a) We know that the *pivot* must occupy index p in the sorted array. We can make use of this fact and only search the section of the array which must contain the element which has the kth index in the sorted array. The pseudocode for quickselect is as follows.

```
function QUICKSELECT(A[0...n-1], k)
    pivot ← SELECTPIVOT(A[0...n-1])
    // p is the index of the pivot in the partitioned array
    p ← PARTITION(A[0...n-1], pivot)
    if p == k then
        return A[k]
    else if p > k then
        return QUICKSELECT(A[0...p-1], k)
    else
        return QUICKSELECT(A[p+1...n-1], k-(p+1))
```

We will assume for this question that we are selecting the first element to be the pivot, but as we have seen in lectures this is not necessarily the best strategy.

(b)

$$\begin{aligned}
\Big[9,3,2,15,10,29,7\Big] &\xrightarrow{\textsc{Partition}} \Big[3,2,7,9,15,10,29\Big],\ p=3<k\\
&k \leftarrow k-(p+1)=0\\
\Big[15,10,29\Big] &\xrightarrow{\textsc{Partition}} \Big[10,15,29\Big],\ p=1>k\\
&k \leftarrow k=0\\
\Big[10\Big] &\xrightarrow{\textsc{Partition}} \Big[10\Big],\ p=0==k\\
&\Longrightarrow \textbf{return } 10
\end{aligned}$$

(c) The best-case for QUICKSELECT is when $p = k$ after the first partition. With our pivot strategy of selecting $pivot = A[0]$ this corresponds to having the kth-smallest element at the start of the array.

The single PARTITION call takes $\Theta(n)$ time, so the best-case time complexity for QUICKSELECT is $\Theta(n)$.

(d) The worst-case for QUICKSELECT is when each call to PARTITION only rules out one element, and thus we have to PARTITION on n elements, $n-1$ elements, $n-2$ elements and so on.

An example of this is finding $k = 0$ when the input is reverse sorted order.

The time taken for each partition is linear, so the total time for QUICKSELECT in the worst case is,

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2).$$

(e) To compute the expected time-complexity of this algorithm we will make the assumption that the order of the array is uniformly random, and that after each PARTITION, the pivot ends up in the middle of the array, *i.e.*, $p = \frac{n}{2}$.

In this case, the recurrence relation for QUICKSELECT will be:

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

We see that this first the format required by the Master theorem with $a = 1, b = 2, c = 1$. Since $log_b(1) = 0 < 1 = c$ we have that $T(n) \in \Theta(n^c) = \Theta(n)$.

Thus QUICKSELECT has an expected time-complexity of $\Theta(n)$.

It turns out that even if the array is split into very unevenly sized components after partitioning (*e.g.,* $0.99n$ and $0.01n$ length sub-arrays) and the kth smallest element *always ends up in the larger array* then the runtime of QUICKSELECT is still $\Theta(n)$. Can you use the Master Theorem to show why this is the case?

(f) Well the worst case for QUICKSELECT is much worse than the worst case for the heap based algorithm ($\Theta(n^2)$ vs. $\Theta(n \log n)$).

Also, if we make the assumption that $k \ll n$ (*i.e.*, k is very small compared to n) then we can treat $\Theta(n + k \log n)$ as $\Theta(n)$ meaning the heap-based approach is comparable to the best case for QUICKSELECT.

In general for unknown k we can expect QUICKSORT to be faster in the expected case.