

CSE3358 Problem Set 4 Solution

Problem 1: Sorting

(a) (10 points) Consider an array A of n numbers each of which is either $\mathbf{0}$ or $\mathbf{1}$. We will refer to such an array as *binary*. Describe an asymptotically efficient algorithm for sorting a binary array A of size n . For full credit you should:

- Describe your algorithm in english
- Provide a clear, neat, and complete pseudocode for your algorithm similar to the way we do it in class
- Analyze correctly the running time of your algorithm and express the running time in either a Θ or an O notation.
- Achieve a running time that is the best possible asymptotically

ANSWER: Since the array contains only zeros and ones, we can count the number of ones by scanning the array and adding the entries. If we find c ones, we can set $A[1] \dots A[n-c]$ to 0 and $A[n-c+1] \dots A[n]$ to 1. Note that this would not work if the array contains objects with keys in $\{0, 1\}$.

Here's a pseudocode for the algorithm.

```
 $c \leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$ 
    do  $c \leftarrow c + A[i]$ 
for  $i \leftarrow 1$  to  $n - c$ 
    do  $A[i] \leftarrow 0$ 
for  $i \leftarrow n - c + 1$  to  $n$ 
    do  $A[i] \leftarrow 1$ 
```

The running time of this algorithm is dominated by the three for loops. Each loop performs a constant amount of work. Therefore, the running time is $\Theta(n) + \Theta(n - c) + \Theta(c) = \Theta(2n) = \Theta(n)$. This is the best asymptotically achievable bound since any sorting algorithm has to read the entire input $A[1] \dots A[n]$ which takes $\Omega(n)$ time.

(b) (10 points) We define a Lomuto Quicksort algorithm as a Quicksort algorithm that (recursively) partitions an array into a left sub-array with all elements \leq Pivot, and a right sub-array with all elements $>$ Pivot.

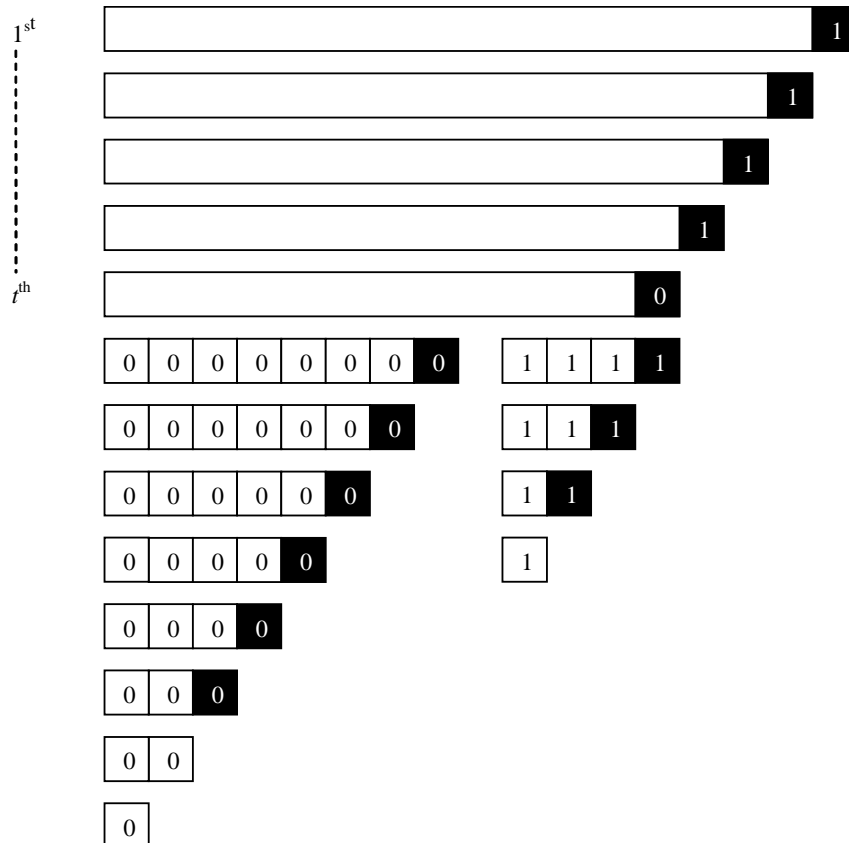
Argue (your argument must be very clear, organized, and convincing, if not a formal proof) that any Lomuto Quicksort algorithm will have an average running time of $\Theta(n^2)$ on binary arrays of size n (i.e. even when pivot is chosen randomly). Explain why this does not contradict the result that QUICKSORT has an average running time of $\Theta(n \log n)$.

ANSWER: The first key observation is that if the pivot is $x = 1$, then the right partition will be empty because there is no element greater than 1. Consider the first time t when a pivot $x = 0$ was chosen. Therefore, the 1st, 2nd, ..., t^{th} partitions are all left partitions where each partition contains one less element than the previous partition. Since the partitioning algorithm takes linear time, we spend $\sum_{i=0}^t \Theta(n - i)$ time on those partitions.

The t^{th} partition with a pivot $x = 0$ produces a left partition L with all zeros and a right partition R with all ones. The second key observation is that partition L will always produce empty right partitions because all elements in L are \leq pivot $x = 0$. Similarly, partition R will always produce empty right partitions because all elements in R are \leq pivot $x = 1$. Pick the largest partition among L and R which will have a size of at least $\lceil \frac{n-t}{2} \rceil = n'$. Since the partitioning algorithm takes linear time, we spend $\sum_{i=0}^{n'} \Omega(n' - i)$.

The total running time will be $\sum_{i=0}^t \Theta(n - i) + \sum_{i=0}^{n'} \Omega(n' - i)$ where $n' = \lceil \frac{n-t}{2} \rceil$. Therefore, the running time $T(n) = \Omega(\sum_{i=0}^t n - i)$ and $T(n) = \Omega(\sum_{i=0}^{n'} n' - i)$. If $t \geq n/2$, then $T(n) = \Omega(\sum_{i=0}^t n - i) = \Omega(\sum_{i=0}^{\lceil n/2 \rceil} n - i) = \Omega(n^2)$. If $t \leq n/2$, then $n' \geq n/4$. Therefore, $T(n) = \Omega(\sum_{i=0}^{n'} n' - i) = \Omega(\sum_{i=0}^{\lceil n/4 \rceil} n) = \Omega(n^2)$. Since $T(n) = O(n^2)$, then $T(n) = \Theta(n^2)$.

Here's an example when the pivot is the last element.



Problem 2: Variations of lists

(1 point each) For each of the following types of lists, what is the asymptotic (Θ notation) worst-case running time of the following dynamic-set operations? (make sure you understand what each operation does).

ANSWER:

	unsorted singly linked	sorted singly linked	unsorted doubly linked	sorted doubly linked
search(L,k)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
insert(L,x)	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
delete(L,x)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
successor(L,x)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
predecessor(L,x)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
minimum(L)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
maximum(L)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Problem 3: Recursion and Stacks

(a) (5 points) Implement in C or C++ the following Quicksort algorithm using the Hoare PARTITION (Hoare is the guy who invented Quicksort).

PARTITION(A, p, r)

$x \leftarrow A[p]$ \triangleright this is the pivot
 $i \leftarrow p-1$ $\triangleright i$ starts at the far left
 $j \leftarrow r+1$ $\triangleright j$ starts at the far right

while TRUE

do repeat $j \leftarrow j-1$ \triangleright move j down until it finds an element $\leq x$
 until $A[j] \leq x$
 repeat $i \leftarrow i+1$ \triangleright move i up until it finds an element $\geq x$
 until $A[i] \geq x$
 if $i < j$ \triangleright if they did not cross
 then swap $A[i] \leftrightarrow A[j]$
 else return j

QUICKSORT(A, p, r)

if $p < r$
 then $q \leftarrow$ PARTITION(A, p, r)
 QUICKSORT(A, p, q) \triangleright note here the difference from the one we saw
 QUICKSORT($A, q+1, r$) \triangleright in class which uses a Lomuto PARTITION

(b) (5 points) Implement in C or C++ a doubly linked list. Each element of the doubly linked list should be able to hold some data. For instance you can use the following declaration:

```

struct Element {
    Element * prev;
    Element * next;
    void * data; //anything you need for the specific application
};

```

ANSWER:

```

struct Element {
    Element * prev;
    Element * next;
    void * data;
};

```

```

struct List {
    Element * head;

    List() {
        head=NULL;
    }
};

```

```

void insert(List * L, Element * x) {
    x->next=L->head;
    if (L->head!=NULL)
        L->head->prev=x;
    L->head=x;
    x->prev=NULL;
}

```

```

void del(List * L, Element * x) {
    if (x->prev!=NULL)
        x->prev->next=x->next;
    else
        L->head=x->next;
    if (x->next!=NULL)
        x->next->prev=x->prev;
}

```

(c) (5 points) Using the linked list of part (b), implement a stack that keeps track of its depth. The stack should provide the following functionality:

- s.push(x): pushes an Element x onto the stack
- s.pop(): pops an Element off the stack and returns it
- s.depth(): returns the number of Elements in the stack
- s.max(): returns the maximum depth ever attained by the stack

ANSWER:

```

class Stack {
    List * L;
    int current_depth;
    int max_depth;

public:

    Stack() {
        L=new List();
        current_depth=0;
        max_depth=0;
    }

    void push(Element * x) {
        current_depth++;
        if (current_depth>max_depth)
            max_depth=current_depth;
        insert(L,x);
    }

    Element * pop() {
        if (current_depth>0) {
            Element * temp=L->head;
            del(L,L->head);
            current_depth--;
            return temp;
        }
        return NULL;
    }

    int depth() {
        return current_depth;
    }

    int max() {
        return max_depth;
    }
};

```

(d) (20 points) Compilers usually execute recursive procedures by using a stack that contains parameter values for each recursive call. The information for the most recent call is at the top of the stack, and the information of the initial call is at the bottom. When a procedure is invoked, its information is pushed onto the stack; when it terminates, its information is popped.

Give a non-recursive implementation in C or C++ of the above Quicksort algorithm using your stack of part (c).

ANSWER: Here's a pseudocode for a non-recursive implementation of Quicksort using a stack.

```

QUICKSORT( $A, p, r$ )
 $S \leftarrow \emptyset$ 
PUSH( $S, (p, r)$ )
while  $S \neq \emptyset$ 
    do  $(p, r) \leftarrow \text{POP}(S)$ 
    if  $p < r$ 
        then  $q \leftarrow \text{PARTITION}(A, p, r)$ 
            PUSH( $A, q + 1, r$ )
            PUSH( $A, p, q$ )

```

Here's an implementation in C++. We need to create a tuple data type to push and pop (p, r) . We will also assume an array of integers.

```

struct Tuple {
    int p;
    int r;
};

Stack s; //outside so you can track its depth

void Quicksort(int * A, int p, int r) {
    Tuple * t=new Tuple();
    Element * e=new Element();
    t->p=p;
    t->r=r;
    e->data=&t;
    s.push(e);
    while (s.depth()) {
        e=s.pop();
        t=(Tuple *) (e->data);
        p=t->p;
        r=t->r;
        if (p<r) {
            q=Partition(A,p,r); //to be implemented
            e=new Element();
            t=new Tuple();
            t->p=q+1;

```

```

    t->r=r;
    e->data=&t;
    s.push(e);
    e=new Element();
    t=new Tuple();
    t->p=p;
    t->r=q;
    e->data=&t;
    s.push(e);
}
}
}

```

Obtain the maximum stack depth for several examples including:

- large sorted array
- large array with all elements equal
- large reverse sorted array

and report the maximum depth of the stack (as returned by `s.max()`) as well as the running time in Θ notation.

ANSWER:

	max. stack depth	running time
sorted array	$\Theta(1)$	$\Theta(n^2)$
array with all elements equal	$\Theta(\log n)$	$\Theta(n \log n)$
reverse sorted array	$\Theta(n)$	$\Theta(n^2)$

(e) (5 points) Can you see that the depth of the stack is $\Theta(n)$ in the worst-case? Which scenarios lead to such a worst-case?

ANSWER: The case of a reverse sorted array. If the order of the two PUSH operations in the code above is changed, the table will be the symmetric one, so the worst-case in terms of stack depth will be the case of a sorted array.

The stack depth that you explicitly observed in part (d) is the stack depth that the compiler uses for recursion. A deep stack means more internal stack operations and more memory usage. It is possible to modify the QUICKSORT code to make the stack depth always $\Theta(\log n)$ even when the worst case occurs, without affection its everage case performance of $\Theta(n \log n)$. We can do this by first reducing the number of recursive calls using a technique called tail recursion When the recursive call is the last instruction in the function it can be eliminated. Here's how:

```

QUICKSORT( $A, p, r$ )
while  $p < r$ 
    do  $q \leftarrow$  PARTITION( $A, p, r$ )
       QUICKSORT( $A, p, q$ )
        $p \leftarrow q + 1$ 

```

(f) (5 points) Argue that this new code for QUICKSORT still correctly sorts the array A .

ANSWER: When the first recursive call returns, all the values for p , r , and q that were before the initiation of the call will be restored. Since p is set to $q+1$, the while loop guarantees that QUICKSORT is executed with arguments $q+1$ and r , which is identical to having another recursive call with these parameters.

(g) (5 points) Describe a scenario where the depth of the internal stack is still $\Theta(n)$ even with the above modification.

ANSWER: The stack depth will be $\Theta(n)$ if there are $\Theta(n)$ recursive calls to QUICKSORT (so this has to be on left partitions now). If the array is sorted, then left partitions will be one element partitions always (try it) and therefore recursion will stop immediately. So we do not go beyond one level of recursion. However, if the array is reverse sorted, we will have $\Theta(n)$ recursive calls on left partitions.

```
QUICKSORT(A,p,r)
QUICKSORT(A,p,r-1)
QUICKSORT(A,p+1,r-2)
QUICKSORT(A,p+2,r-3)
```

...

Therefore, we have almost $n/2$ recursive calls. To see this, try to QUICKSORT a small array, say $A = [8, 7, 6, 5, 4, 3, 2, 1]$.

(h) (5 points) How can you further modify the code to make the stack depth $\Theta(\log n)$ without affecting the average case performance of QUICKSORT? *Hint:* since we use recursion on one part of the array only, why not use it on the smaller part?

ANSWER: The problem demonstrated by the above scenario is that each invocation of QUICKSORT calls QUICKSORT again with almost the same range. To avoid such behavior, we must change QUICKSORT so that the recursive call is on a smaller interval of the array. The following variation of QUICKSORT checks which of the two subarrays returned from PARTITION is smaller and recurses on the smaller subarray, which is at most half the size of the current array. Since the array size is reduced by at least half on each recursive call, the number of recursive calls, and hence the stack depth, is $\Theta(\log n)$ in the worst case. The expected running time is not affected, because exactly the same work is done as before: the same partitions are produced, and the same subarrays are sorted.

```
QUICKSORT(A, p, r)
while p < r
  do q ← PARTITION(A, p, r)
  if q - p + 1 < r - q
    then QUICKSORT(A, p, q)
    p ← q + 1
  else QUICKSORT(A, q + 1, r)
    r ← q
```

Problem 4: Building a Heap

Consider the problem of building a heap from an array A . As we saw in class, this can be done by repeated calls to HEAPIFY.

```
BUILD-HEAP( $A$ )
  for  $i \leftarrow \lfloor n/2 \rfloor$  downto 1
    do HEAPIFY( $A, i$ )
```

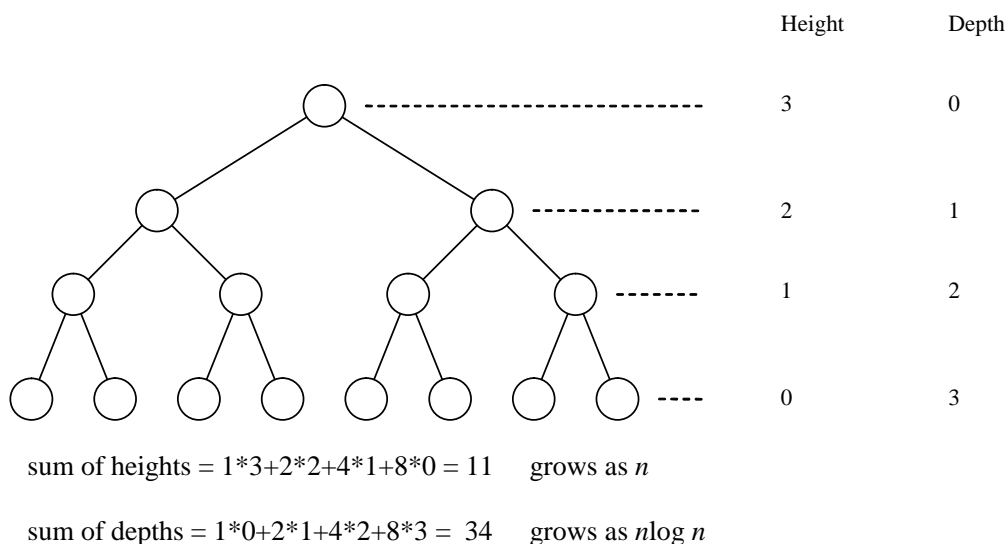
(a) (5 points) Suggest another *inplace* algorithm for building a heap and provide its pseudocode and analyze its running time.

ANSWER: One possible way of building a Heap is to repeatedly insert the elements one by one.

Note: We have been treating n , the size of the heap, as a global variable in our pseudocode.

```
BUILD-HEAP( $A$ )
   $m \leftarrow n$ 
   $n \leftarrow 0$ 
  for  $i \leftarrow 1$  to  $m$ 
    do HEAP-INSERT( $A, A[i]$ )
```

Each HEAP-INSERT operation takes $O(\log n)$ time, where n is the eventual size of the heap after all insertions. Therefore, the running time of this algorithm is $O(n \log n)$. Note that it is not possible with a tight analysis of the running time of n consecutive inserts to obtain an $O(n)$ time bound as in the case of n consecutive HEAPIFY operations. In fact, it can be shown that n consecutive inserts take $\Theta(n \log n)$ time. The difference is that the running time of BUILD-HEAP with HEAPIFY is proportional to the sum of heights which is $\Theta(n)$, but the running time of BUILD-HEAP with inserts is proportional to the sum of depths, which is $\Theta(n \log n)$. Here's an visual illustration:



Another way of building a heap (inplace) is to sort A using any inplace sorting algorithm like Insertionsort ($\Theta(n^2)$ time) or Quicksort ($\Theta(n \log n)$ time on average).

(b) (5 points) Does your algorithm produce the same result as the one based on repeated HEAPIFY? If yes, provide an argument. If no, provide a counter example.

ANSWER: Not necessarily. Consider the array $A = [1, 2, 3]$. BUILD-HEAP with HEAPIFY will produce $A = [3, 2, 1]$. BUILD-HEAP with inserts will produce $A = [3, 1, 2]$.

(c) (5 points) Given n distinct elements, how many different heaps of these elements can you build (I have no idea now)? How does this number change with n asymptotically?

ANSWER: Let $H(n)$ =number of heaps of n distinct elements. Given a heap of n elements, the root is always the maximum element. Therefore, we have to consider variations on the remaining $n - 1$ elements. Let x be the number of elements in the root's left subtree. The left subtree is also a heap; therefore, the left subtree is a heap of x elements. Similarly, the right subtree is a heap of $n - 1 - x$ elements. Since the heap is a nearly complete binary tree, x can always be determined from n , and for a given n , x is fixed. The x elements in the left subtree can be any x elements among the $n - 1$ elements. Therefore, we can write the expression of $H(n)$ as follows:

$$H(n) = C_x^{n-1} H(x) H(n - 1 - x)$$

Let's not worry about how to compute x and focus on complete heaps only. In a complete heap, $x = \frac{n-1}{2} = \lfloor n/2 \rfloor$. Therefore, for $n = 2^p - 1$ for some $p \geq 0$, we have the following formula:

$$H(n) = C_{\lfloor n/2 \rfloor}^{n-1} H^2(\lfloor n/2 \rfloor)$$

Using the expression for $C_{\lfloor n/2 \rfloor}^{n-1} = \frac{(n-1)!}{\lfloor n/2 \rfloor! \lfloor n/2 \rfloor!}$ and Stirling's approximation for $n! = \Theta(\frac{n^{n+1/2}}{e^n})$, we get $C_{\lfloor n/2 \rfloor}^{n-1} = \Theta(\frac{2^n}{\sqrt{n}})$. Let $h(n) = \log H(n)$. Then (ignoring floors)

$$h(n) = \log C_{\lfloor n/2 \rfloor}^{n-1} + 2h(n/2) = \log(\Theta(\frac{2^n}{\sqrt{n}})) + 2h(n/2) = \Theta(n) + 2h(n/2)$$

Using the Master theorem, $h(n) = \Theta(n \log n)$. Therefore, $H(n) = 2^{h(n)} = n^{\Theta(n)}$. Here's a table showing the number of heaps for $n = 0, 1, \dots, 20$.

n	#heaps
0	1
1	1
2	1
3	2
4	3
5	8
6	15
7	80
8	210

9	896
10	2520
11	19200
12	69300
13	506880
14	2059200
15	21964800
16	108108000
17	820019200
18	3920716800
19	48881664000
20	279351072000