CHRISTOFER FABIAN CHÁVEZ CARAZAS

1. Problems

2. Getting Started

2.1. Insert-Sort

2.1.1.

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array A = {31,41,59,26,41,58}

- 31,**41**,59,26,41,68 (The key is 41. This remains in place).
- 31,41,**59**,26,41,68 (The key is 59. This remains in place).
- 31,41,59,**26**,41,58 (The key is 26. This takes the position of the number 31).
- 26,31,41,49,**41**,58 (The key is 41. This takes the position of the number 59).
- 26,31,41,41,59,**58** (The key is 58. This takes the position of the number 59).
- **Result:** 26,31,41,41,58,59

2.1.2.

Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.

```
void insertSort(vector<int> &vec){
    for(int j = 1; j < vec.size(); j++){
        int key = vec[j];
        int i = j - 1;
        while(i >= 0 and vec[i] < key){
            swap(vec[i + 1], vec[i]);
            i--;
        }
    }
}
```

2.1.3.

Consider the **searching problem:**
Input: A sequence of i such that v = A[i] or the special value *NIL* if v does not appear in A.

Write pseudocode for **linear search**, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

```
for (int j = 0; i < A.size(); j++){
  if(v == A[j]) return j;
}
return NULL;
```

2.1.4.

Cosider the problem of adding two *n-bit* binary integers, stored in two *n-element* arrays A and B. The sum of the two integers should be stored in binaty form in an (n+1)-elemnt array C. State the problem formally and write pseudocode for adding the two integers.

```
vector<int> sumBin(vector<int> &A, vector<int> &B){
    int n = A.size();
    vector<int> C(n + 1);
    int l = 0;
    for(int i = n - 1; i >= 0; i--){
        if(!A[i] and !B[i]){
            C[i + 1]  = l;
            l = 0;
        }
        else if(A[i] and B[i]){
            C[i + 1] = l;
            if(!l) l = !l;
        }
        else{
            if((A[i] or B[i]) and l){
                C[i + 1] = 0;
                l = 1;
            }
            else C[i + 1] = 1;
        }
    }
    C[0] = l;
    return C;
}
```

2.2. Analyzing algorithms

2.2.1.

Express the function $n^3/1000 - 100n^2 + 3$ in terms of Θ-notation

$$n^3/1000 - 100n^2 - 100n + 3 < n^3 - 100n^3 - 100n^3 + 3n^3$$
$$n^3/1000 - 100n^2 - 100n + 3 < 203n^3$$
$$\Theta(n^3)$$

2.2.2.

Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. The find the second smallest element of a , and exchange it with $A[2]$. Continue in this manner for the first $n-1$ element of A. Write pseudocode for this algorithm, which is know as **selection sort**.

- **What lop invariant does this algorithm mainthain?**
- **Why does it need to run only the first $n-1$ rather than for all n elements?** Because allways the last element is in the correct site.
- **Give the best-case and worst-case running times of selection sort int Θ-notation.**

```
SELECTION SORT                                     cost   times
1 $n$ = A.length                                   $C1$   $1$
2 for $i = 0$ to $n-1$                             $C2$   $n$
3     $menor$ = NUMERO INFINITAMENTE GRANDE         $C3$   $n-1$
4     $index = 0$                                  $C4$   $n-1$
5     for $j = i$ to $n$                           $C5$   $\sum_{i=1}^{n+1} i$
6         if $A[j] < menor$ then                   $C6$   $\sum_{i=0}^{n} i$
7             $menor = A[j]$                       $C7$   $\sum_{i=0}^{n} i$
8             $index = j$                          $C8$   $\sum_{i=0}^{n} i$
9     swap($A[index]$,$A[i]$)                      $C9$   $n-1$
```

$T(n) = C1 + nC2 + C3(n-1) + C4(n-1) + C5(\sum_{i=1}^{n+1} i)$
$+ C6(\sum_{i=0}^{n} i) + C7(\sum_{i=0}^{n} i) + C8(\sum_{i=0}^{n} i) + C9(n-1)$

$T(n) = C1 + nC2 + nC3 + nC4 + nC9 - C3 - C4 - C9 + C5(\frac{n^2+3n+2}{2})$
$+ C6(\frac{n^2+n}{2}) + C7(\frac{n^2+n}{2}) + C8(\frac{n^2+n}{2})$

$T(n) = n^2(\frac{C6+C7+C8+C5}{2}) + n(C2 + C3 + C4 + C9 + \frac{3C5+C6+C7+C8}{2})$
$+ C1 - C3 - C4 - C9$

$T(n) = An^2 + Bn + C$

$T(n) \in \Theta(n^2)$

2.2.3.

Consider linear search again.

- **How many elemnts of the input sequence need to be checked on the average, assuming that the element begin searched for is equally likely to be any element in the array?**

- **How about in the worst case?**
- **What are the average-case and worts-case running times of linear searching in Θ-notation?**

LINEAR SEARCH	cost	times
1 for $i = 0$ to A.length	$c1$	$n + 1$
2 if($A[i] == v$) return i	$c2$	n
3 return NIL	$c3$	1

$T(n) = c1(n + 1) + c2(n) + c3$
$T(n) = n(c1 + c2) + c3 + c1$
$T(n) = An + B$
$T(n) \in \Theta(n)$

2.2.4.

How can we modify almost any algorithm to have a good best-case running time?

It can improve the algorithm efficiency if the data input is efficient too.

2.3. Designing algorithms

2.3.1.

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = 3, 41, 52, 26, 38, 57, 9, 49$.

First Sequence

3 41 52 26 38 57 9 49

Second Sequence

3 41 26 52 38 57 9 49

Third Sequence

3 26 41 52 9 38 49 57

Sorted Sequence

3 9 26 38 41 49 52 57

2.3.2.

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

2.3.3.

Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$T(n) = \begin{cases} 2 & if \;\; n = 2, \\ 2T(n/2) + n & if \;\; n = 2^k, for \;\; k > 1 \end{cases}$

is $T(n) = n \lg n$

$T(n) = 2T(n/2) + n$
$T(n) = 2(2T(n/4) + n) + n$
$T(n) = 4T(n/4) + 3n$
$T(n) = 4(2T(n/8) + n)3n$
$T(n) = 8(T(n/8) + 7n$
$\vdots$
$T(n) = \frac{n}{2}(T(\frac{n}{n/2})) + n(\frac{n}{2} - 1)$
$T(n) = \frac{n}{2}(2) + n(\frac{n}{2} - 1)$
$T(n) = n + \frac{n^2}{2} - n$
$T(n) = \frac{n^2}{2}$

2.3.4.

We can express insertion sort as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

$T(n) = \begin{cases} 1 & if \;\; n = 1 \\ T(n-1) + n & if \;\; n > 1 \end{cases}$

$T(n) = T(n-1) + n$
$T(n) = n + (n-1) + T(n-2)$
$T(n) = n + (n-1) + (n-2) + T(n-3)$
$T(n) = n + (n-1) + (n-2) + \cdots + 2 + T(1)$
$T(n) = \frac{n(n-1)}{2} + 1$
$T(n) \in \Theta(n^2)$

2.3.5.

Referring back to the searching problem, observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from futher consideration. The **binary search** algorithm repeats this procedure, halving the size of the remaining portion for the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

```
BINARY SEARCH                  cost   times
1 Do
2     media = $\frac{i+f}{2}$           $c1$     $m$
2     if $(A[medio] == v)$     $c2$     $m$
3         return true              $c3$     1
4     if $(A[medio] > v)$       $c4$     $m$
5         $f = medio + 1$          $c5$     $m$
6     else $i = M + 1$           $c5$     $m$
7 while $i <= f$                   $c6$     $m$
```

$T(n) = m(c1 + c2 + c4 + c5 + c6) + c3$
$T(n) = Am + B$
$n = 2^m$
$m = \lg m$
$T(n) = A \lg n + B$
$T(n) \in \Theta(\lg n)$

2.3.6.

Observe that the **while** loop of lines 5-7 of lines 5-7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) though the sorted subarray $A[1..j-1]$. Can we use a binary search instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

2.4. Problems

2.4.1. Insertion sort on small arrays in merge sort

1. **Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time**.

$L = n/k$	Where L is the number of sublists
$\Theta(k^2)$	Time delay algorith insert sort to sort a sublist.
$\Theta(k^2 * L)$	Multiplying for L
$\Theta(\cancel{k^{\cancel{2}}} * \frac{n}{\cancel{k}})$	By solving the equation.
$\Theta(nk)$	Result

2. **Show hoe to merge the sublist in $\Theta(n \lg(n/k))$ worst-case time.**

   The original merge runs in $\Theta(n \lg n)$, the n into the lg represents the most sublists that can divide. In this case, the most sublists that can divide is n/k, consequently the merge in this problem runs in $\Theta(n \log(n/k))$ worst-case time.

3. **Given that the modified algorithm runs in $\Theta(nk+n \log(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified**

algorithm has the same running time as standard merge sort, in terms of Θ-notation

If we take a $k = 1$ then, the modified algorithm raid in $\Theta(n \log n)$

4. **How should we choose k in practice?**

   Choose k most tiny.

2.4.2. Correctness of bubblesort

1. **Let A' denote the output of BUBBLESORT(A). To peove that BUBBLESORT is correct, we need to prove that it terminates and that**

   $$A'[1] \leq A'[2] \leq \cdots \leq A'[n]$$

   where $n = A.kength$. In order to show that BUBBLESORT actually sorts, what else do need to prove?

   I need prove that the loop of the bubblesort is invariant.

2. **State precisely a loop invariant for the for loop in lines 2-4, and prove that this invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.**

3. **Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the for loop in lines 1-4 that will allow you to prove inequality. Your proof should use the structure of the loop invariant proof presented in this chapter.**

4. **What is the worst-case running time of bubblesort? How does it compare to the running time of onsertion sort?**

   The bubblesort runs in $\Theta(n^2)$ worst-case time. The difference with the insert sort is that the first have a bigger constant.

2.4.3. Correctness of Horner's rule

1. **In terms of Θ-notation, what is the running time of this code fragment for Horner's rule?**

   Is $\Theta(n)$

2. **Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each terms of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?**

2.4.4. Inversion

1. **List the five Inversion of the array $\{2,3,8,6,1\}$**
   - 1,3,8,6,2
   - 2,1,8,6,3
   - 2,3,6,8,1
   - 2,3,1,6,8
   - 2,3,8,1,6
2. **What array with elements from the set $\{1,2,....n\}$ has the most inversions? How many does it have?**

   Haven't any
3. **What is the relationship between the running time of insertion sort and be number of inversions in the input array?**
4. **Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time.**

```
MERGE-INVERSIONS(A,p,q,r)
1 $n_1 = q - p + 1$
2 $n_2 = r - q$
3 let $L[1..n_1 + 1] and R[1..n_2 + 1]$ be new arrays
4 for $i = 1$ to $n_1$ 5      $L[i] = A[p + i - 1]$
6 for $j = 1$ to $n_2$ 7      $R[J] = A[q + j]$
8 $L[n_1 + 1] = \infty$
9 $R[n_2 + 1] = \infty$
10 $i = 1$
11 $j = 1$
12 $invertions = 0$
13 for $k = p$ to $r$
14      if $L[I] \leq R[j]$
15           $A[k] = L[i]$
16           $i = i + 1$
17      else $A[k] = R[j]$
18           $j = j + 1$
19           $invertions++$
20 return $invertions$
```

```
MERGE-SORT-INVERSIONS(A,p,r)
1 $inversions = 0$
2 if $p < r$
3      $q = [(p + r)/2]$
4      $inversions += $ MERGE-SORT-INVERSIONS(A,p,q)
```

```
5    $inversions += $ MERGE-SORT-INVERSIONS(A,q +1,r)
6    $inversions += $ MERGE-INVERSIONS(A,p,q,r)
7 return $inversions$
```

3. Growth of Functions

3.1. Asymptotic notation

3.1.1.

Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $max(f(n), g(n)) = \Theta(f(n) + g(n)))$.

3.1.2.

Show that for any real constants a and b, where $b > 0$,

$$(n + a)^b = \Theta(n^b)$$

$$\begin{gathered}
(n + a)^b \leq Cn^b \\
\frac{(n+a)^b}{n^b} \leq C \\
(\frac{n+a}{n})^b \leq C \\
(1 + \frac{a}{n})^b \leq C \\
\text{Min value} = 1 \\
\text{Max value} = (1 + a)^b \\
\Rightarrow C_1 = (1 + a)^b \land C_2 = 1
\end{gathered}$$

3.1.3.

Explain why the statement, The running time of algorithm A is at least $O(n^2)$, is meaningless.

3.1.4.

$$\begin{array}{cc}
\text{Is } 2^{n+1} = O(2^n)? & \text{Is } 2^{2n} = O(2^n)? \\
2^{n+1} \leq C(2^n) & 2^{2n} \leq C(2^n) \\
\frac{2^{n+1}}{2^n} \leq C & \frac{2^{2n}}{2^n} \leq C \\
\frac{\cancel{2^n}*2}{\cancel{2^n}} \leq C & (\frac{2^{\cancel{2}}}{\cancel{2}})^n \leq C \\
2 \leq C & 2 \leq C
\end{array}$$