CISC 3220 Robert Wagner
Assignment 1 September 4, 2016

1 Exercises 1.1

1. Give a real-world example that requires sorting or a real-world example that requires computing a convex hull.

   Bank transaction database

2. Other than speed, what other measures of efficiency might we use in a real-world setting?

   Minimize: Communcation data, programmer development time
   Maximize: User-friendliness

3. Select a data structure you have used previously. Describe its strengths and weaknesses.

   Linked List:
   Strengths: Dynamic size, Constant time insert + delete + concatenate
   Weaknesses: Sequential (non-random) access, not-compact (pointer overhead)

4. How are the shortest path and traveling salesman problems similar? How do they differ?

   Similar: They are both graph traversal optimization problems
   Different: In the TS problem the begin and endpoint are fixed at the same point, In the SP problem the endpoints are free to vary.

5. Come up with a real-world problem in which only the best solution will do, then come up with a situation in which an approximation is enough.

   Best only: Wolfram-Alpha style algebraic solver
   Approximate: Almost all real-world problems ... Floating point error makes almost all computations approximate to some degree.

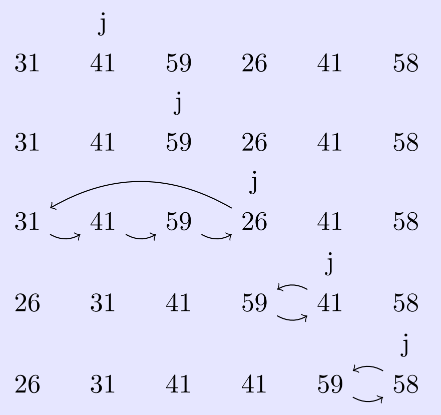
2 Problem 1-1 Comparison of running times

For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

	1s	1m	1h	1d	1M	1Y	1C
$\log_2 n$	10^{301030}	6×10^{301031}	3.6×10^{301033}	8.6×10^{301034}	2.6×10^{301036}	!!!	!!!
$\sqrt{n}$	10^{12}	6.0×10^{13}	3.6×10^{15}	8.6×10^{16}	2.6×10^{18}	3.2×10^{19}	3.2×10^{21}
n	10^{6}	6.0×10^{7}	3.6×10^{9}	8.6×10^{10}	2.6×10^{12}	3.2×10^{13}	3.2×10^{15}
$n \log_2 n$	6.4×10^{4}	3.8×10^{6}	2.3×10^{8}	5.5×10^{9}	1.7×10^{11}	2.0×10^{12}	2.0×10^{14}
n^2	1000	6.0×10^{4}	3.6×10^{6}	8.6×10^{7}	2.6×10^{9}	3.2×10^{10}	3.2×10^{12}
n^3	100	6.0×10^{3}	3.6×10^{5}	8.6×10^{6}	2.6×10^{8}	3.2×10^{9}	3.2×10^{11}
2^n	20	1.2×10^{3}	7.2×10^{4}	1.7×10^{6}	5.1×10^{7}	6.2×10^{8}	6.2×10^{10}
$n!$	9	540	3.2×10^{4}	7.7×10^{5}	2.3×10^{7}	2.8×10^{8}	2.8×10^{10}

3 Exercises 2.1

1. Using figure 2.2 as a model, illustrate the operation of insertion sort on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.



```
      j
31    41    59    26    41    58
            j
31    41    59    26    41    58
                  j
31    41    59    26    41    58
                        j
26    31    41    59    41    58
                              j
26    31    41    41    59    58
```

Final sorted output:

```
26    31    41    41    58    59
```

2. Rewrite the insertion sort to sort into non-ascending order instead of non-descending.

```
for j := 2 to A.length
    key := A[j]
    i := j - 1
    while i > 0 and A[i] < key
        A[i + 1] = A[i]
        i := i - 1
    A[i + 1] := key
```

3. Consider the searching problem.
   input: a sequence of numbers $A = \langle a_1, a_2, \cdots, a_n \rangle$ and a value v.
   output: an index i such that $v = A[i]$ or the special value NIL if $v \notin A$.
   Write pseudo-code for linear search. Use a loop invariant to prove your algorithm is correct.

```
FIND(A, v)
    for i := 1 to A.length
        if v = A[i]
            return i
    return NIL
```

Loop invariant: for each index value i in the loop, we define a subset $A_i \subseteq A$ such that $A_i = A[1 \cdots i]$.
Initial: At the beginning of each loop iteration, we know that $v \notin A_{i-1}$, where $A_0 = \emptyset$.
Maintenance: If $v = a_i$ then we terminate the loop immediately.
Otherwise, since $A_i = A_{i-1} \cup \{a_i\}$, $v \notin A_{i-1}$ and $v \notin \{a_i\}$ it follows that $v \notin A_i$ and the invariant holds.
Termination: Either the loop terminates when $v = a_i$ invalidates the invariant, or when $i > A$.length.

4. Consider the problem of adding two n-bit binary integers stored in two n-bit arrays A and B. The sum of the two integers should be stored in binary form in an $(n + 1)$ element array C. State the problem formally and write pseudo-code.

There are two components to a binary sum: Remainder term that keeps its current place (A XOR B) and a carried bit (A AND B) that is shifted to the next higher place (B SHL 1). Loop invariant: $A + B = k$ where k is constant. I actually return A here to cover the case where B=0 to start, since A + 0 = A.

```
while B != 0
    C = A XOR B
    B = (A AND B) SHL 1
    A = C
return A
```

4 Exercises 2.2

1. Express the function $\frac{n^3}{1000} - 100n^2 - 100n + 3$ in terms of Θ notation.

$\Theta(n^3)$

2. Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$, then find the second smallest and exchange it with $A[2]$, and continue for the first $(n - 1)$ elements of A. Write pseude-code for this algorithm known as selection sort. Why does it need to run for only $(n - 1)$ elements? Give best-case and worst-case running time on Θ notation.

```
for i := 1 to n-1
    key := A[i]
    index := i
    for j := i + 1 to n
        if A[j] < key
            key := A[j]
            index := j
    A[index] := A[i]
    A[i] := key
```

Selection sort outer loop only runs to $(n - 1)$ since the loop invariant guarantees that all elements in the subarray $A'[1 \cdots i]$ are less than all elements in the subarray $A[i + 1 \cdots n]$. Thus when the $(n - 1)$ iteration has terminated, the element $A[n] \geq A'[k]$ for all $k \in \{1 \cdots n - 1\}$, which means an nth iteration of the outer loop is unnecessary.
Worst-case time: $\Theta(n^2)$, Best-case time: $\Theta(n^2)$.

3. Consider linear search again. How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times? Justify your answer.

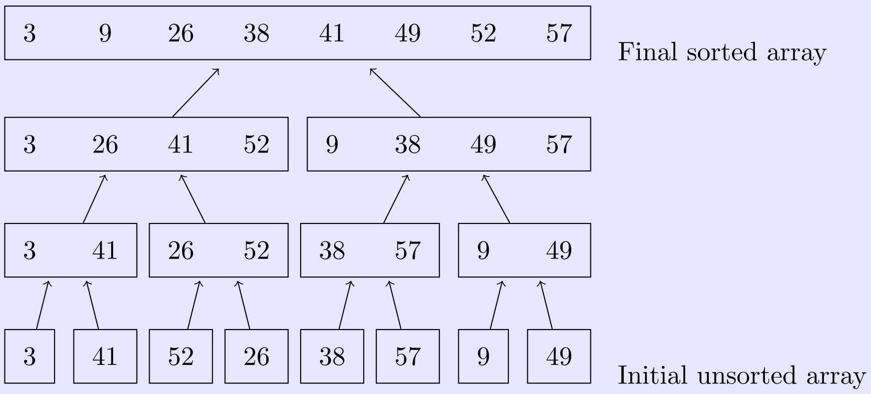
On average we need to check $n/2$ elements, and in the worst case we need to check all n elements. The running time is thus $\Theta(n/2)$ in the average case and $\Theta(n)$ in the worst case.

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

By tailoring the algorithm to optimize the most frequently encountered input set, and short-circuiting execution by testing for best-case input and returning early.

5 Exercises 2.3

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



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

```
MERGE(A, p, q, r)
    n1 := q - p + 1
    n2 := r - q
    let L[1 ..  n1] and R[1 ..  n2] be new arrays
    for i := 1 to n1
        L[i] := A[p + i - 1]
    for j := 1 to n2
        R[j] := A[q + j]
    i := j := 1
    k := p
    while k <= p and i <= n1 and j <= n2
        if L[i] <= R[j]
            A[k++] := L[i++]
        else A[k++] := R[j++]
    while i <= n1
        A[k++] := L[i++]
    while j <= n2
        A[k++] := R[j++]
```

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

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

is $T(n) = n \log_2 n$.

Base Case: Let $k = 1$, thus

$$n = 2^k = 2^1 = 2$$
$$T(2^k) = T(2) = 2\log_2 2 = (2)(1) = 2 ✓$$

Inductive Step: Let $k > 1$, and suppose $n = 2^k$. Note that

$$\begin{aligned} n/2 &= 2^{k-1} \\ T(n/2) &= T(2^{k-1}) \\ k &= \log_2 n \end{aligned}$$

Assume $T(2^{k-1}) = 2^{k-1} \log_2 2^{k-1}$ is true. Then

$$T(2^k) = T(n) = 2(2^{k-1} \log_2 2^{k-1}) + 2^k$$

Which simplifies to

$$\begin{aligned} T(2^k) = T(n) &= (2^k)(k - 1 + 1) = (2^k)(k) \\ &= n \log_2 n \end{aligned}$$

Thus we have shown that the inductive step holds and the recurrence is true for all $k > 1$. □

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

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

Where $T(n-1)$ term refers to the recursive sort $A[1..n-1]$ and $\Theta(n)$ refers to the time to insert $A[n]$ into the sorted array.

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 the sequence from consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portions of the sequence each time. Write pseude-code for binary search. Argue that the worst-case running time is $\Theta(\log_2 n)$.

```
BSEARCH(A, l, r, v)
    mid := l + (l - r) / 2
    if A[mid] = v
        return mid
    if mid = l
        return NIL
    if v < A[mid]
        return BSEARCH(A, l, mid - 1, v)
    else
        return BSEARCH(A, mid + 1, r, v)
```

Note that on each recursive iteration, we discard half of the previous iteration set. Thus our worst-case recurrence is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n/2) + \Theta(1) & \text{if } n > 1 \end{cases}$$

Which can be shown to be $T(n) = \log_2 n$ by the recursion tree method. Thus the worst-case running time is $\Theta(\log_2 n)$.

6. Observe that the while loop of lines 5-7 of the insertion sort procedure in section 2.1 uses a linear search to scan backwards through the sorted sub-array. Can we use a binary search to improve the worst-case running time to $\Theta(n \log_2 n)$?

NO, using the binary search will not improve the worst-case time since the algorithm has to insert the a_j element into its place in the already sorted A_{j-1} subarray, which requires $\Theta(j-1)$ worst-case time per iteration of j, which sums to $\Theta(n^2) >> \Theta(n \log_2 n)$.

7. Describe an $\Theta(n \log_2 n)$ algorithm that, given a set S of n integers, and another integer x, determines whether there exists two numbers in S whose sum is x.

First, Sort the set S into an ordered list S' which requires $\Theta(n \log_2 n)$ worst-case time.
Second, Iterate through S' where $i = 1 \cdots n-1$ indexes the element s'_i.
On each iteration, do a binary search through the subarray of S' to the right of the current index for a value $v = x - S'[i]$. If the value is found we have fulfilled our test and can terminate the loop. This process requires n iterations of $\Theta(\log_2 n) = \Theta(n \log_2 n)$ in the worst case.
Thus the worst-case time is $\Theta(n \log_2 n) + \Theta(n \log_2 n) = \Theta(n \log_2 n)$.

6 Problem 2-1 Insertion sort on small arrays in merge sort

Although merge sort runs in $\Theta(n \log_2 n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Consider a modification of merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merge mechanism, where k is a value to be determined.

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

For each of the sublists, the size is k thus the worst-case running time of insertion sort is $\Theta(k^2)$ per sublist. Since there are n/k of them, the total worst-case time is $(n/k)\Theta(k^2)$ which cancels out to $\Theta(nk)$.

2. Show how to merge the sublists in $\Theta(n \log_2(n/k))$ worst-case time.

$$T(n) = \begin{cases} \Theta(k) & \text{if } n \le k \\ 2T(n/2) + \Theta(1) & \text{if } n > k \end{cases}$$

Since the recurrence stops when $n = k$, this results in
$T(n) = [(n/k)\Theta(k)] \times [\Theta(log_2 n) - \Theta(log_2 k)] = \Theta(n \log_2(n/k))$

3. Given that the modified algorithm runs in $\Theta(nk + n \log_2(n/k))$ 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?

$$\begin{aligned} \Theta(nk + \log_2(n/k)) &\le \Theta(n \log_2 n) \\ nk + log_2 n - log_2 k &\le n log_2 n \\ nk - log_2 k &\le (n-1) log_2 n \\ k &\le \frac{n-1}{n} \log_2 n + C \end{aligned}$$

4. How should we choose k in practice?

Suppose $n = 2^x$ for some $x \in \mathbb{R}$. Let $k = \lfloor x \rfloor$. This is easy to calculate and guarantees that $k \le \log_2 n$.

7 Problem 2-2 Correctness of Bubblesort

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```
BUBBLESORT(A)
1    for i := 1 to A.length - 1
2        for j := A.length downto i + 1
3            if A[j] < A[j-1]
4                exchange A[j] with A[j-1]
```

1. Let A' denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that
$$A'[1] \leq A'[2] \leq \cdots \leq A'[n]$$
where $n = A$.length. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

   That A' is a permutation of A, i.e. we can define a bijection $f : A \to A'$.

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

   Loop invariant: the lowest value element of subarray $S = A[j \cdots n]$ (where $n = A$.length) is $A[j]$.
   Initialization: On first iteration $j = n$ so S has a single element and $A[j]$ is the lowest value trivially.
   Maintenance: On each iteration, if $A[j-1]$ is a lower value than $A[j]$ then they are exchaged, so that the loop invariant holds for the next iteration when j is decremented.
   Termination: The loop terminates when $j = i + 1$, and the loop invariant guarantees that $A[i]$ is the lowest value element of the subarray $A[i \cdots n]$, i.e. $A[i] \leq A[k]$ for all $k \in \{i + 1 \cdots n\}$.

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 the inequality above.

   Loop invariant: The subarray $T = A'[1 \cdots i]$ is sorted, i.e. $A'[1] \leq \cdots \leq A'[i]$.
   Initialization: On first iteration $i = 1$ so T has a single element and is sorted trivially.
   Maintenance: On each iteration, The inner loop guarantees $A'[i]$ is less than or equal to all elements that follow. Thus all elements before i were the lowest value of the subarray on their respective iteration, and thus the invariant holds.
   Termination: The loop terminates when the subarray has one element, and the array is sorted.

4. What is the worst-case running time of bubblesort? how does it compare to the running time of insertion sort?

   The worst case running time is $\Theta(n^2)$ which is identical to the worst case of Insertion sort. In practice however insertion sort is faster than bubble sort since it executes far fewer exchanges in the average case, even though it executes a similar number of compares.

8 Problem 2-3 Correctness of Horner's Rule

The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)\cdots)) \end{aligned}$$

Given the coefficients $a_0, a_1, \cdots a_n$ and a value for x:

```
y = 0
for i = n downto 0
    y = a_i + xy
```

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

   $\Theta(n+1)$ where n is the degree of the polynomial.

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

   ```
   total := 0
   for k := 1 to n
       term := A[k]
       for j := 1 to k
           term := term * x
       total := total + term
   return total
   ```
   The worst-case running time of this algorithm is $\Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$, which is worse than Horner's.

3. Consider the following loop invariant:
   At the start of each iteration of the for loop of lines 2-3,

   $$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

   Interpret a summation with no terms equalling 0. Following the structure of the loop invariant proof presented in this chapter, use the loop invariant to show that, at termination,

   $$y = \sum_{k=0}^{n} a_k x^k$$

   Initialization: before the first iteration

   $$y = 0 = \sum_{k=0}^{-1} a_{n+1} x^0$$

   since the sum bounds are invalid, the result is 0 which is consistent.

   Maintenance:

   $$y' = a_i + xy = a_i x^0 + \sum_{k=0}^{n-i-1} a_{k+i+1} x^{k+1}$$

If we define $k' = k + 1$ and $i' = i - 1$, then

$$y' = a_i x^0 + \sum_{k'=1}^{n-(i'+1)} a_{k'+(i'+1)} x^{k'}$$

We can combine the $a_i x^0$ term into the sum as a $k' = 0$ term

$$y' = \sum_{k'=0}^{n-(i'+1)} a_{k'+i'+1} x^{k'}$$

Which maintains the loop invariant for the next iteration since i is decremented each iteration.

Termination: At the termination of our final iteration, $i' = -1$ and

$$y' = \sum_{k'=0}^{n-(-1+1)} a_{k'+1-1} x^{k'} = \sum_{k=0}^{n} a_k x^k$$

4. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by coefficients $a_0, a_1, \cdots, a_n$.

We have shown that a loop invariant is maintained which terminates in a state where the value $y = \sum_{k=0}^{n} a_k x^k$ which is the value a polynomial with coefficients $a_0, a_1, \cdots, a_n$ evaluates to. Thus the code fragment correctly evaluates a polynomial of finite degree.