O Notation 3

Simple Sorting

Simple Sorting

- As an example of algorithm analysis let's look at two simple sorting algorithms
  - Selection Sort and
  - Insertion Sort
- Calculate an approximate cost function for these two sorting algorithms
  - By analyzing how many operations are performed by each algorithm
  - This will include an analysis of how many times the algorithms' loops iterate

Selection Sort

- The array is divided into sorted part and unsorted parts
- Expand the sorted part by swapping the first unsorted element with the smallest unsorted element
  - Starting with the element with index 0, and
  - Ending with the last but one element (index $n - 1$)
- Requires two processes
  - Finding the smallest element of a sub-array
  - Swapping two elements of the array

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
arr	1	3	5	7	21	27	29	23	19	13	31	15	17	9	25	11

smallest: 9

The algorithm is on its fifth iteration

Find the smallest element in arr[4:15]

Selection Sort

- The array is divided into sorted part and unsorted parts
- Expand the sorted part by swapping the first unsorted element with the smallest unsorted element
  - Starting with the element with index 0, and
  - Ending with the last but one element (index $n - 1$)
- Requires two processes
  - Finding the smallest element of a sub-array
  - Swapping two elements of the array

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
arr	1	3	5	7	9	27	29	23	19	13	31	15	17	21	25	11

smallest: 9

The algorithm is on its fifth iteration

Find the smallest element in arr[4:15]

Swap smallest and first unsorted elements

Selection Sort Algorithm

```
void selectionSort(int arr[], int n){
    for(int i = 0; i < n-1; ++i){
        int smallest = getSmallest(arr, i, n);
        swap(arr, i, smallest);
    }
}
```

```
int getSmallest(int arr[], int start, int end){
    int smallest = start;
    for(int i = start + 1; i < end; ++i){
        if(arr[i] < arr[smallest]){
            smallest = i;
        }
    }
    return smallest;
}
```

note: end is 1 past the last legal index

```
void swap(int arr[], int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

Selection Sort Analysis – Swap

```
void selectionSort(int arr[], int n){
    for(int i = 0; i < n-1; ++i){
        int smallest = getSmallest(arr, i, n);
        swap(arr, i, smallest);
    }
}
```

n-1 swaps

3(*n*-1)

```
int getSmallest(int arr[], int start, int end){
    int smallest = start;
    for(int i = start + 1; i < end; ++i){
        if(arr[i] < arr[smallest]){
            smallest = i;
        }
    }
    return smallest;
}
```

Swap always performs three operations

```
void swap(int arr[], int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

Selection Sort Analysis – Smallest

```
void selectionSort(int arr[], int n){
     for(int i = 0; i < n-1; ++i){
          int smallest = getSmallest(arr, i, n);
          swap(arr, i, smallest);
     }
}
```

called n-1 times

n-1 swaps

3(n-1)

```
int getSmallest(int arr[], int start, int end){
     int smallest = start;
     for(int i = start + 1; i < end; ++i){
          if(arr[i] < arr[smallest]){
               smallest = i;
          }
     }
     return smallest;
}
```

4(*end*-*start*-1)+4 operations

n is passed the value of *end*

start is passed the value of i_{ss}

i_{ss} increases in the *ss* for loop

i_{ss} sequence = {0,1,…,n-3,n-2}

i_{gs} = start + 1 = i_{ss} + 1

i_{gs} sequence = {1,2,…,n-2,n-1}

for_{gs} iterations = {n-1,n-2,…,2,1}

average = ?

Counting Digression

- It is common to find loops that iterate over all the elements of a data structure
  - That are nested inside some other loop
- We may need to find
  - The sum of the iterations of the inner loop
  - And the average of the iterations of the inner loop
- The average is easy
  - Sum of inner iterations / number of outer iterations
  - How do we derive the sum of the inner iterations?

```
foo(int n){
    for(…) { //outer
        for(…) { //inner
            body
        }
    }
}
```

When *foo* is called how many times is *body* executed?

iterations of outer
*
average iterations of inner

Counting Sequences

- Linear loops are sequences where subsequent values increase or decrease by one
  - 1,2,3,4,5,6,7,8,9,10 sum? 55 ((1+10)/2) * 10)
  - 24,25,26,27,28,29,30,31,32,33,34,35 sum? 354 ((24+35)/2) * 12)
  - 1,2,…,n-2,n-1 sum? $n*(n-1)/2 = (n^2-n)/2$ ((1+n-1)/2 * n-1)
- Sum of a sequence *s*, of length *n* is $((s_0 + s_{n-1}) / 2) * n$
- Consider this demonstration – not a proof
  - Sum the sequence 1 : *n*-1 twice, then divide by 2

sequence	1	2	3	…	n-3	n-2	n-1
sequence	n-1	n-2	n-3	…	3	2	1
sum	**n**	**n**	**n**	**n**	**n**	**n**	**n**

Average? n/2

Selection Sort Analysis – Smallest

```
void selectionSort(int arr[], int n){
    for(int i = 0; i < n-1; ++i){
        int smallest = getSmallest(arr, i, n);
        swap(arr, i, smallest);
    }
}
```

called n-1 times

n-1 swaps

3(n-1)

```
int getSmallest(int arr[], int start, int end){
    int smallest = start;
    for(int i = start + 1; i < end; ++i){
        if(arr[i] < arr[smallest]){
            smallest = i;
        }
    }
    return smallest;
}
```

4(*start*-*end*-1)+4 operations

n is passed the value of *end*

start is passed the value of i_{ss}

i_{ss} increases in the *ss* for loop

i_{ss} sequence = {0,1,…,n-3,n-2}

i_{gs} = start + 1 = i_{ss} + 1

i_{gs} sequence = {1,2,…,n-2,n-1}

for_{gs} iterations = {n-1,n-2,…,2,1}

average = n/2

cost = 4(n/2)+4

Selection Sort Analysis – Smallest

```
void selectionSort(int arr[], int n){
    for(int i = 0; i < n-1; ++i){
        int smallest = getSmallest(arr, i, n);
        swap(arr, i, smallest);
    }
}
```

called n-1 times

(n-1)(4(n/2)+4)

= (n-1)(2n + 4)

= $2n^2$-2n+4(n-1)

= $2n^2$-2n+4n-4

= $2n^2$+2n-4

n-1 swaps

3(n-1)

3n-3

for loop: 3(n-1)+2

3n-1

Cost function: $t_{\text{selection sort}} = 2n^2 + 8n - 8$

Barometer Operation

- The barometer operation for selection sort is in the loop that finds the smallest item
  - Since operations in that loop are executed the greatest number of times
- The loop contains four operations
  - Compare *i* to *end*
  - Compare *arr*[*i*] to smallest
  - Change smallest
  - Increment *i*

The barometer instructions

```
int getSmallest(arr[], start, end)
      smallest = start
      for(i = start + 1; i < end; ++i)
            if(arr[i] < arr[smallest])
                  smallest = i
      return smallest
```

Barometer Operations

Unsorted elements	Barometer
n	n-1
n-1	n-2
…	…
3	2
2	1
1	0
	$n(n-1)/2$

Selection Sort Cases

- How is selection sort affected by the organization of the input?
  - The only work that varies based on the input organization is whether or not smallest is assigned the value of *arr*[*i*]
- What is the worst-case organization?
- What is the best-case organization?
- The difference between best case and worst case is quite small
  - $(n-1)(3(n/2)) + 10(n-1) + 2$ in the best case and
  - $(n-1)(4(n/2)) + 10(n-1) + 2$ in the worst case

Selection Sort Summary

- Ignoring leading constants, selection sort performs the following work
  - $n*(n-1)/2$ barometer operations, regardless of the original order of the input
  - $n-1$ swaps
- The number of comparisons dominates the number of swaps
- The organization of the input only affects the leading constant of the barometer operations

Insertion Sort

- The array is divided into sorted part and unsorted parts
- The sorted part is expanded one element at a time
  - By moving elements in the sorted part up one position until the correct position for the first unsorted element is found
    - Note that the first unsorted element is stored so that it is not lost when it is written over by this process
  - The first unsorted element is then copied to the insertion point

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
arr	5	11	27	31	21	1	29	23	19	13	7	15	17	9	25	3

temp: 21

The algorithm is on its fourth iteration

Find the correct position for arr[4]

Insertion Sort

- The array is divided into sorted part and unsorted parts
- The sorted part is expanded one element at a time
  - By moving elements in the sorted part up one position until the correct position for the first unsorted element is found
    - Note that the first unsorted element's value is stored so that it is not lost when it is written over by this process
  - The first unsorted element is then copied to the insertion point

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
arr	5	11	21	27	31	1	29	23	19	13	7	15	17	9	25	3

temp: 21

The algorithm is on its fourth iteration

Find the correct position for arr[4]

Move up elements in the sorted part until the position for 21 is found

Work Performed in Inserting Values

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
arr	5	11	21	27	31	1	29	23	19	13	7	15	17	9	25	3

- How much work was performed by expanding the sorted part of the array by one element?
  - The value 21 was stored in a variable, *temp*
  - The values 27 and 31 were compared to 21
    - And moved up one position in the array
  - The value 11 was compared to 21, but not moved
  - The value of *temp* was written to *arr*[2]
- How much work will be performed expanding the sorted part of the array to include the value 1?
- How much work will be performed expanding the sorted part of the array to include the value 29?

Insertion Sort Algorithm

```
void insertionSort(int arr[], int n){
    for(int i = 1; i < n; ++i){
        temp = arr[i];
        int pos = i;
        // Shuffle up all sorted items > arr[i]
        while(pos > 0 && arr[pos - 1] > temp){
            arr[pos] = arr[pos – 1];
            pos--;
        } //while
        // Insert the current item
        arr[pos] = temp;
    }
}
```

What are the barometer operations?

How often are they performed?

It depends on the values in the array

Insertion Sort Algorithm

```
void insertionSort(int arr[], int n){
    for(int i = 1; i < n; ++i){
        temp = arr[i];
        int pos = i;
        // Shuffle up all sorted items > arr[i]
        while(pos > 0 && arr[pos - 1] > temp){
            arr[pos] = arr[pos – 1];
            pos--;
        } //while
        // Insert the current item
        arr[pos] = temp;
    }
}
```

outer loop
n-1 times

inner loop body
how many times?

worst case: *pos* – 1 times for each iteration

pos ranges from 1 to *n*-1; *n*/2 on average

What is the worst-case organization?

outer loop runs *n*-1 times: $n * (n - 1) / 2$

Insertion Sort Worst Case Cost

Sorted Elements	Worst-case Search	Worst-case Move
0	0	0
1	1	1
2	2	2
…	…	…
n-1	n-1	n-1
	$n(n-1)/2$	$n(n-1)/2$

Insertion Sort Worst Case

- In the worst case the array is in reverse order
- Every item has to be moved all the way to the front of the array
  - The outer loop runs n-1 times
    - In the first iteration, one comparison and move
    - In the last iteration, n-1 comparisons and moves
    - On average, n/2 comparisons and moves
  - For a total of $n * (n-1) / 2$ comparisons and moves

Insertion Sort Best Case

- The efficiency of insertion sort *is* affected by the state of the array to be sorted
- What is the best case?
  - In the best case the array is already completely sorted!
  - No movement of any array element is required
  - Requires *n* comparisons

Insertion Sort: Average Case

- What is the average case cost?
  - Is it closer to the best case?
  - Or the worst case?
- If *random* data is sorted, insertion sort is usually closer to the worst case
  - Around $n * (n\text{-}1) / 4$ comparisons
- And what do we mean by average input for a sorting algorithm anyway?