Searching and sorting

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Hamilton

Last updated: January 22, 2025

Linear search, binary search, selection sort, insertion sort

Notions related to program correctness

- Soundness: is the output always as expected?

Notions related to program correctness

- Soundness: is the output always as expected?
  - if the program produced output, then the output is correct

Notions related to program correctness

- Soundness: is the output always as expected?
  - if the program produced output, then the output is correct
- Completeness: does the program always produce an output?

Notions related to program correctness

- Soundness: is the output always as expected?
  - if the program produced output, then the output is correct
- Completeness: does the program always produce an output?
  - if there exists a solution, then the program will produce an output

Notions related to program correctness

- Soundness: is the output always as expected?
  - if the program produced output, then the output is correct
- Completeness: does the program always produce an output?
  - if there exists a solution, then the program will produce an output
- Correct: sound and complete

Notions related to program correctness

- Soundness: is the output always as expected?
  - if the program produced output, then the output is correct
- Completeness: does the program always produce an output?
  - if there exists a solution, then the program will produce an output
- Correct: sound and complete
- Partially correct: sound but not complete

Notions related to program correctness

- Soundness: is the output always as expected?
  - if the program produced output, then the output is correct
- Completeness: does the program always produce an output?
  - if there exists a solution, then the program will produce an output
- Correct: sound and complete
- Partially correct: sound but not complete
  - program may not halt on some inputs

Notions related to program correctness

- Soundness: is the output always as expected?
  - if the program produced output, then the output is correct
- Completeness: does the program always produce an output?
  - if there exists a solution, then the program will produce an output
- Correct: sound and complete
- Partially correct: sound but not complete
  - program may not halt on some inputs
- Loop invariant: conditions that are true before the loop and after every iteration

Linear search: Interface

```c
typedef int ItemType;

//
// Helper function: linearSearch
//
// Purpose: Locate the first occurrence of x in the array A.
// Parameter(s):
//  <1> x: An ItemType item to be sought.
//  <2> A: An array of ItemType in which the search
//     is to be conducted.
//  <3> n: An unsigned integer indicating the scope of the search.
// Precondition(s): N/A
// Returns: If x occurs in A[0:n], then the index of
//    the first occurrence will be returned.
// Otherwise, -1 will be returned.
// Side Effect: N/A
```

Linear search: Implementation

```
int linearSearch(const ItemType x,
                 const ItemType A[],
                 unsigned int n) {
  for (unsigned int i = 0; i < n; i++){
    if (x == A[i]){
      return i;
    }
  }
  return -1;
}
```

- Time complexity: as the name suggests, linear

Linear search: Implementation

```
int linearSearch(const ItemType x,
                 const ItemType A[],
                 unsigned int n) {
  for (unsigned int i = 0; i < n; i++){
    if (x == A[i]){
      return i;
    }
  }
  return -1;
}
```

- Time complexity: as the name suggests, linear
  - searching through n elements takes time proportional to n

Linear search: Implementation

```
int linearSearch(const ItemType x,
                 const ItemType A[],
                 unsigned int n) {
  for (unsigned int i = 0; i < n; i++){
    if (x == A[i]){
      return i;
    }
  }
  return -1;
}
```

- Time complexity: as the name suggests, linear
  - searching through n elements takes time proportional to n
  - Twice as many elements -> twice as much time

Binary search: Idea

- Works correctly on sorted data only

Binary search: Idea

- Works correctly on sorted data only
  - Will find some occurrence of searched item x (may not be the first one)

Binary search: Idea

- Works correctly on sorted data only
    - Will find some occurrence of searched item x (may not be the first one)
- Check the middle item m

Binary search: Idea

- Works correctly on sorted data only
  - Will find some occurrence of searched item `x` (may not be the first one)
- Check the middle item `m`
  - if `x == m`, we have found `x`

Binary search: Idea

- Works correctly on sorted data only
  - Will find some occurrence of searched item `x` (may not be the first one)
- Check the middle item m
  - if `x == m`, we have found `x`
  - if `x < m` then `x` will not be located to the right of `m`, and thus `x` should be sought for in the subarray to the left of `m`

Binary search: Idea

- Works correctly on sorted data only
  - Will find some occurrence of searched item `x` (may not be the first one)
- Check the middle item m
  - if `x == m`, we have found `x`
  - if `x < m` then `x` will not be located to the right of `m`, and thus `x` should be sought for in the subarray to the left of `m`
  - if `x > m` then `x` will not be located to the left of `m`, and thus `x` should be sought for in the subarray to the right of `m`

Interface

```
//
// binarySearch
//
// Purpose: To determine if an array contains the specified element.
// Parameter(s):
//  <1> x: The element to search for
//  <2> A: The array to search in
//  <3> n: The length of array A
// Precondition(s): N/A
// Returns: Whether element x is in array A.
// Side Effect: N/A
```

Implementation

```
bool binarySearch(ItemType x, const ItemType A[], unsigned int n) {
  /*1*/ int low = 0;
  /*2*/ int high = n - 1;

  /*3*/ while (low <= high) {
    /*4*/ int mid = (low + high) / 2;
    /*5*/ if (x == A[mid])
      /*6*/ return true;
    /*7*/ else if (x < A[mid])
      /*8*/ high = mid - 1;
    /*9*/ else
      /*10*/ low = mid + 1;
  } // end while
  /*11*/ return false;
}
```

Time Complexity

- If the array holds 32 items, needs roughly 5 steps

Time Complexity

- If the array holds 32 items, needs roughly 5 steps
- If the array holds 2048 items, needs roughly 11 steps

Time Complexity

- If the array holds 32 items, needs roughly 5 steps
- If the array holds 2048 items, needs roughly 11 steps
    - why?

Time Complexity

- If the array holds 32 items, needs roughly 5 steps
- If the array holds 2048 items, needs roughly 11 steps
    - why?
- In general, in the worst case, at most $\log_2(n) + 1$ steps

Time Complexity

- If the array holds 32 items, needs roughly 5 steps
- If the array holds 2048 items, needs roughly 11 steps
  - why?
- In general, in the worst case, at most $\log_2(n) + 1$ steps
  - Twice as many items => only one extra step

Time Complexity

- If the array holds 32 items, needs roughly 5 steps
- If the array holds 2048 items, needs roughly 11 steps
  - why?
- In general, in the worst case, at most $\log_2(n) + 1$ steps
  - Twice as many items => only one extra step
- Let's analyze the case for 4 items

Time Complexity

- If the array holds 32 items, needs roughly 5 steps
- If the array holds 2048 items, needs roughly 11 steps
  - why?
- In general, in the worst case, at most $\log_2(n) + 1$ steps
  - Twice as many items => only one extra step
- Let's analyze the case for 4 items
- How about 7 items?

Sorting: Definition

- Rearranging items in some sort of order (either ascending or descending)

Sorting: Definition

- Rearranging items in some sort of order (either ascending or descending)
- useful for many applications

Sorting: Definition

- Rearranging items in some sort of order (either ascending or descending)
- useful for many applications
- many known sorting algorithms exist: selection sort, insertion sort, bubble sort, quick sort, merge sort, heap sort, shell sort, radix sort, etc.

Sorting: Definition

- Rearranging items in some sort of order (either ascending or descending)
- useful for many applications
- many known sorting algorithms exist: selection sort, insertion sort, bubble sort, quick sort, merge sort, heap sort, shell sort, radix sort, etc.
- each have different performance characteristics (e.g., quick sort is the fastest in the average case, while heap sort and merge sort are the fastest in the worst case)

The selection sort algorithm: Idea

- The minimum member of the original array will be the first element of the sorted array

The selection sort algorithm: Idea

- The minimum member of the original array will be the first element of the sorted array
- If we take away the the first element, then the minimum element of the remaining subarray will be the second element in the sorted order

The selection sort algorithm: Idea

- The minimum member of the original array will be the first element of the sorted array
- If we take away the the first element, then the minimum element of the remaining subarray will be the second element in the sorted order
- If we take away the second element, then the minimum element of the remaining subarray will be the third element in the sorted order

The selection sort algorithm: Idea

- The minimum member of the original array will be the first element of the sorted array
- If we take away the the first element, then the minimum element of the remaining subarray will be the second element in the sorted order
- If we take away the second element, then the minimum element of the remaining subarray will be the third element in the sorted order
- … so on and so forth

The selection sort algorithm: Idea

- The minimum member of the original array will be the first element of the sorted array
- If we take away the the first element, then the minimum element of the remaining subarray will be the second element in the sorted order
- If we take away the second element, then the minimum element of the remaining subarray will be the third element in the sorted order
- … so on and so forth
- So, repeatedly select the minimum element from the remaining elements and places it next in the ordering, until all elements have been ordered

The selection sort algorithm: Idea

- The minimum member of the original array will be the first element of the sorted array
- If we take away the the first element, then the minimum element of the remaining subarray will be the second element in the sorted order
- If we take away the second element, then the minimum element of the remaining subarray will be the third element in the sorted order
- … so on and so forth
- So, repeatedly select the minimum element from the remaining elements and places it next in the ordering, until all elements have been ordered
- Example using 2 arrays?

Two Array Pseudocode

- Sort array A[n]:

```
for (i = 0; i < n; i++){
```

```
}
```

Two Array Pseudocode

- Sort array A[n]:

```
for (i = 0; i < n; i++){
```

- 1. find the min element in the unsorted array

```
}
```

Two Array Pseudocode

- Sort array A[n]:

```
for (i = 0; i < n; i++){
```

- 1. find the min element in the unsorted array
- 2. remove min element from unsorted array

```
}
```

Two Array Pseudocode

- Sort array A[n]:

```
for (i = 0; i < n; i++){
```

- 1. find the min element in the unsorted array
- 2. remove min element from unsorted array
- 3. place min element at index i of sorted array

```
}
```

One Array Pseudocode

- Sort array A[n]:

```
for (i = 0; i < n; i++){



}
```

One Array Pseudocode

- Sort array A[n]:

```
for (i = 0; i < n; i++){
```

- 1. find the min element in the unsorted region of array A

```
}
```

One Array Pseudocode

- Sort array A[n]:

```
for (i = 0; i < n; i++){
```

- 1. find the min element in the unsorted region of array A
- 2. swap the min element with the element at index i

```
}
```

Loop Invariants

- Recall loop invariants: at the end of each iteration i

```
for (i = 0; i < n; i++){

}
```

Loop Invariants

- Recall loop invariants: at the end of each iteration i
  - the subarray A[0..i-1] is a prefix of the sorted array

```
for (i = 0; i < n; i++){




}
```

Loop Invariants

- Recall loop invariants: at the end of each iteration i
    - the subarray A[0..i-1] is a prefix of the sorted array
    - the subarray A[i..n] contains the remaining elements in some arbitrary order

```
for (i = 0; i < n; i++){



}
```

Loop Invariants

- Recall loop invariants: at the end of each iteration i
  - the subarray A[0..i-1] is a prefix of the sorted array
  - the subarray A[i..n] contains the remaining elements in some arbitrary order
- Refined version:

```
for (i = 0; i < n; i++){
```

```
}
```

Loop Invariants

- Recall loop invariants: at the end of each iteration i
  - the subarray A[0..i-1] is a prefix of the sorted array
  - the subarray A[i..n] contains the remaining elements in some arbitrary order
- Refined version:

```
for (i = 0; i < n; i++){
```

- 1. find the min element in A[i..n]

```
}
```

Loop Invariants

- Recall loop invariants: at the end of each iteration i
  - the subarray A[0..i-1] is a prefix of the sorted array
  - the subarray A[i..n] contains the remaining elements in some arbitrary order
- Refined version:

```
for (i = 0; i < n; i++){
```

- 1. find the min element in A[i..n]
- 2. swap the min element with A[i]

```
}
```

Implementation

```
void selectionSort(ItemType A[], unsigned int n){
  for (unsigned int i = 0; i < n; i++){
    unsigned int m = min(A, i, n);
    swap(A[i], A[m]);
  }
}
```

Min Helper Function

```
unsigned int min(const ItemType A[],
                 unsigned int begin,
                 unsigned int end){
  assert(begin <= end);
  unsigned int m = begin;
  for (unsigned int i = begin + 1; i < end; i++){
    if (A[m] > A[i])
      m = i;
  }
  return m;
}
```

Swap Helper Function

```
void swap(ItemType &x, ItemType &y) {
  ItemType tmp = x;
  x = y;
  y = tmp;
}
```

Another Implementation

```
void selectionSort(ItemType A[], int N){
  int i, j, search_min;
  ItemType temp;

  for (i = 0; i < N; i++) {
    // Find index of smallest element
    search_min = i;
    for (j = i + 1; j < N; j++) {
      if (A[j] < A[search_min])
        search_min = j;
    }
    // Swap items
    temp = A[search_min];
    A[search_min] = A[i];
    A[i] = temp;
  } // end for
}
```

The Insertion Sort algorithm

- Divide the unsorted array into two regions

The Insertion Sort algorithm

- Divide the unsorted array into two regions
  - sorted “left” region/subarray

The Insertion Sort algorithm

- Divide the unsorted array into two regions
  - sorted “left” region/subarray
  - unsorted “right” region/subarray

The Insertion Sort algorithm

- Divide the unsorted array into two regions
  - sorted “left” region/subarray
  - unsorted “right” region/subarray
- Incrementally take one element from the unsorted region

The Insertion Sort algorithm

- Divide the unsorted array into two regions
    - sorted “left” region/subarray
    - unsorted “right” region/subarray
- Incrementally take one element from the unsorted region
    - insert it into the sorted region to generate a sorted region that is one element larger

The Insertion Sort algorithm

- Divide the unsorted array into two regions
  - sorted “left” region/subarray
  - unsorted “right” region/subarray
- Incrementally take one element from the unsorted region
  - insert it into the sorted region to generate a sorted region that is one element larger
- Rinse and repeat

The Insertion Sort algorithm

- Divide the unsorted array into two regions
  - sorted “left” region/subarray
  - unsorted “right” region/subarray
- Incrementally take one element from the unsorted region
  - insert it into the sorted region to generate a sorted region that is one element larger
- Rinse and repeat
- Sorting happens when inserting element (and not when selecting it)

Intertion Sort Pseudocode

- Sort A[n]:

```
for i ranging from 0 to n-1 do {
    Select x = A[i];
    Insert x into sorted region on the left;
  }
```

Intertion Sort Pseudocode

- Sort A[n]:

```
for i ranging from 0 to n-1 do {
    Select x = A[i];
    Insert x into sorted region on the left;
  }
```

- Example?

Invariant

- At the end of each iteration i:

```
for i ranging from 0 to n-1 do {
    Select x = A[i];
    Insert x into subarray A[0..i];
  }
```

Invariant

- At the end of each iteration i:
    - the subarray A[0..i] is sorted,

```
for i ranging from 0 to n-1 do {
    Select x = A[i];
    Insert x into subarray A[0..i];
  }
```

Invariant

- At the end of each iteration i:
    - the subarray A[0..i] is sorted,
    - while the subarray A[i+1..n] is in some arbitrary order

```
for i ranging from 0 to n-1 do {
    Select x = A[i];
    Insert x into subarray A[0..i];
  }
```

Invariant

- At the end of each iteration i:
  - the subarray A[0..i] is sorted,
  - while the subarray A[i+1..n] is in some arbitrary order
- Sort A[n]:

```
for i ranging from 0 to n-1 do {
    Select x = A[i];
    Insert x into subarray A[0..i];
  }
```

Implementation

```
void insertionSort(ItemType A[], unsigned int n) {
  for (unsigned int i = 0; i < n; i++) {
    ItemType x = A[i];
    // Find insertion point
    unsigned int j = find(x, A, i);
    // Shift elements
    shiftRight(A, j, i);
    // Store element
    A[j] = x;
  }
}
```

Helper Function: Find

```
unsigned int find(ItemType x, const ItemType A[], unsigned int n) {
  for (unsigned int i = 0; i < n; i++) {
    if (A[i] >= x)
      return i;
  }
  return n;
}
```

Helper Function: shiftRight

```
void shiftRight(ItemType A[], unsigned int begin, unsigned int end) {
  assert(0 <= begin);
  assert(begin <= end);

  for (unsigned int j = end; j > begin; j--)
    A[j] = A[j-1];

}
```

Another Implementation

```
void insertionSort(ItemType A[], int N) {
  int i, j, insert_index;
  ItemType x;

  for (int i = 0; i < N; i++) {
    // save the element from position i
    x = A[i];

    // Find the insertion point
    insert_index = 0;
    while ((insert_index < i) && (x > A[insert_index]))
      insert_index++;
    // Shift the elements
    for (j = i; j > insert_index; j--)
      A[j] = A[j-1];

    // Store x at the insertion point
    A[insert_index] = x;
  }
}
```

Bonus: Bubble Sort

- Main idea:

Bonus: Bubble Sort

- Main idea:
    - Repeatedly go through array

Bonus: Bubble Sort

- Main idea:
  - Repeatedly go through array
  - Look at side-by-side elements

Bonus: Bubble Sort

- Main idea:
    - Repeatedly go through array
    - Look at side-by-side elements
    - If the left one is bigger, swap them

Bonus: Bubble Sort

- Main idea:
  - Repeatedly go through array
  - Look at side-by-side elements
  - If the left one is bigger, swap them
- Can do with two nested loops

Bonus: Bubble Sort

- Main idea:
  - Repeatedly go through array
  - Look at side-by-side elements
  - If the left one is bigger, swap them
- Can do with two nested loops
  - After outer loop's run `i`, the `i` largest elements are sorted at end of the array

Bonus: Bubble Sort

- Main idea:
  - Repeatedly go through array
  - Look at side-by-side elements
  - If the left one is bigger, swap them
- Can do with two nested loops
  - After outer loop's run `i`, the `i` largest elements are sorted at end of the array
  - After each inner loop's `j` run, the `i` th largest element is not in the first `j` elements

Bonus: Bubble Sort

- Main idea:
  - Repeatedly go through array
  - Look at side-by-side elements
  - If the left one is bigger, swap them
- Can do with two nested loops
  - After outer loop's run `i`, the `i` largest elements are sorted at end of the array
  - After each inner loop's `j` run, the `i` th largest element is not in the first `j` elements
- See: `https://www.youtube.com/watch?v=37E3wokWzlU`

Bubble Sort Code

```
typedef int ItemType;

void bubbleSort(ItemType A[], int N){
  for (int i = 0; i < N-1; i++){
    for (int j = 0; j < (N-1)-i; j++){
      if (A[j] > A[j+1]){
        swap(A[j], A[j+1]);
      }}}}

int main(){
  int A[10] = {2, 3, 5, 4, 1, 4, 99, 3000, 0, -33};
  bubbleSort(A, 10);
  for (int i = 0; i < 10; i++){
    cout << A[i] << " ";
  } cout << endl;
}
```

```
-33 0 1 2 3 4 4 5 99 3000
```