

Thoughts on Project 4

Comparison Sorts I

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, September 28, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

© 2005–2020 Glenn G. Chappell

Some material contributed by Chris Hartman

Review

Part 1

A different kind of **search**: searching for solutions to a problem.

During such a search, we might need to **backtrack**—undo work in order to try something different.

Backtracking search works well when we have a notion of a **partial solution**. A partial solution that is finished is a **full solution**.

It is often convenient to implement backtracking using recursion.

- A recursive call means “Look for full solutions based on this partial solution.”
- Return means “backtrack”.

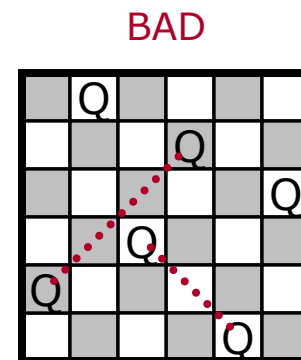
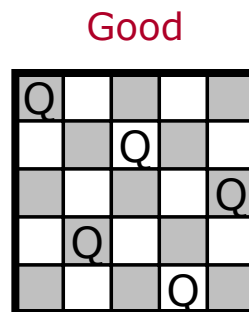
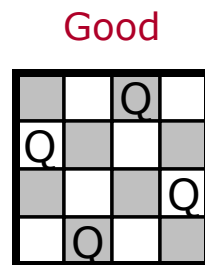
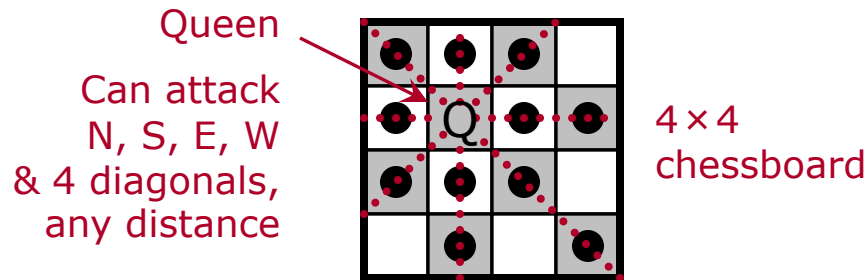
Review

Recursive Backtracking [2/4]

We wrote code to print solutions to the ***n*-Queens Problem**.

- Place n queens on an $n \times n$ chessboard so that none of them can attack each other.

See `nqueen.cpp`.

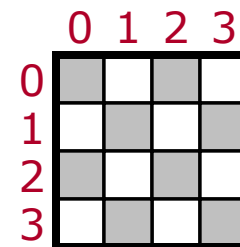
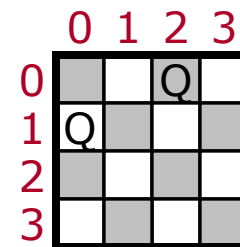
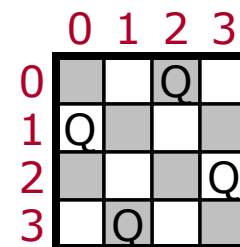


A **partial solution** is a non-attacking placement of queens on the first 0 or more rows of the board.

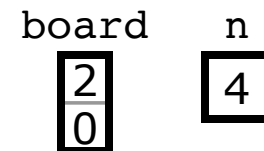
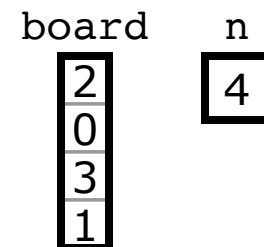
Representing a Partial Solution

- Number rows and columns 0 .. $n-1$.
- Two variables:
 - `board` (vector of `int`).
 - `n` (`int`).
- Variable `n` holds the number of rows/columns in a full solution.
- Variable `board` holds the columns of queens already placed, one per row.
- The size of variable `board` is the number of rows in which queens have been placed.

Partial
Solution




Representation



We can also **count** solutions. Each recursive call returns the number of full solutions based on a given partial solution.

The Code

See `nqueencount.cpp`.

- **Nonrecursive wrapper function**
 - Create an empty partial solution.
 - Call the workhorse function with this partial solution.
 - Return the return value of the workhorse function.
- **Recursive workhorse function** is given a partial solution, returns the number of full solutions that can be made from it.
 - Do we have a full solution?
 - If so, then return 1.
 - Do we have a clear dead end?  **This might be unnecessary.**
 - If so, then return 0.
 - Otherwise:
 - Set *total* to zero.
 - For each way of extending the current partial solution, make a recursive call, and add its return value to *total*.
 - Return *total*.

Thoughts on Project 4

Thoughts on Project 4

Introduction

Now we look at the problem you are to solve in Project 4: counting “holey spider walks”.

You will write code to solve this problem using recursive backtracking.

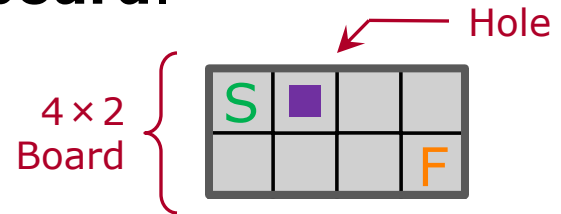
Thoughts on Project 4

Problem Description [1/4]

Consider a rectangle divided into equal-size squares. One square is labeled **start**; another square is labeled **finish**. A third square is marked as a **hole**. We call the result a **board**.

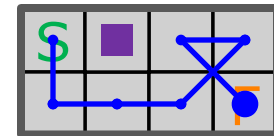
What happens:

- Place a “**spider**” on the **start** square.
- The **spider** can step north, south, east, west, or the four diagonal directions, to an adjacent board square, but not to the **hole**.
- We want the spider to walk in this way around the board, stepping on each square, except the **hole**, exactly once, and ending on the **finish** square.



A path that accomplishes this is a **holey spider walk**.

Here is an example of a holey spider walk on the above board.

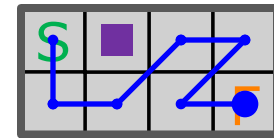
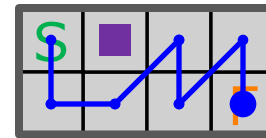
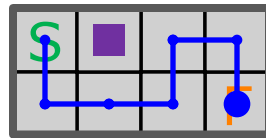
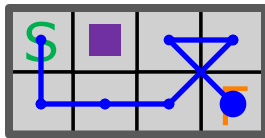
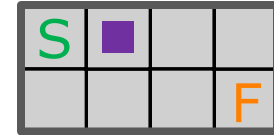


Thoughts on Project 4

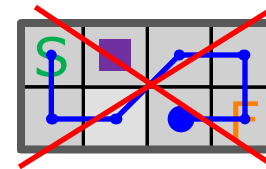
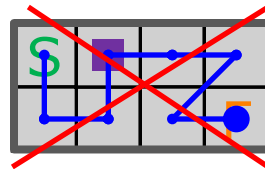
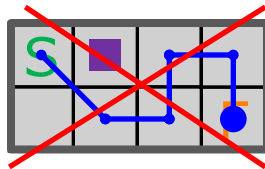
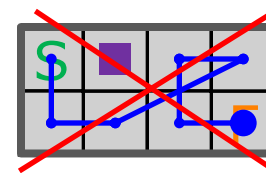
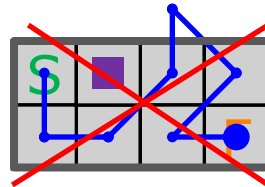
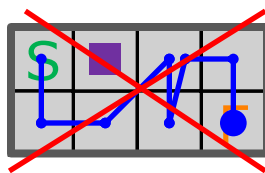
Problem Description [2/4]

How many different holey spider walks does this board have?

The answer turns out to be 4. Here they are.



Here are some paths that are *not* holey spider walks for this board.



Thoughts on Project 4

Problem Description [3/4]

Place a coordinate system on a board, starting in the upper-left corner, as shown. Specify squares as x, y .

- On our 4×2 board, the hole is at $(1, 0)$.

	0	1	2	3
0	S			
1				F

Your job in Project 4 is to write a function `countHSW` that takes the board dimensions, **hole** location, and **start** and **finish** squares, each specified as x, y , and returns the number of holey spider walks on that board.

So `countHSW(4, 2, 1, 0, 0, 0, 3, 1)` should return 4.

dimensions hole start finish

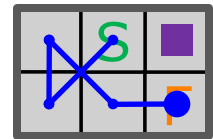
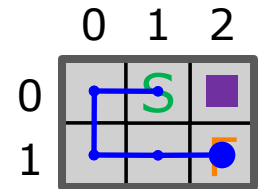
Thoughts on Project 4

Problem Description [4/4]

Many other boards are possible.

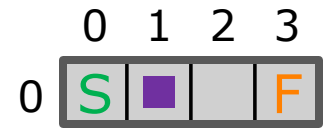
Here is a 3×2 board. It has exactly 2 holey spider walks; both are shown.

- `countHSW(3,2, 2,0, 1,0, 2,1)` returns 2.



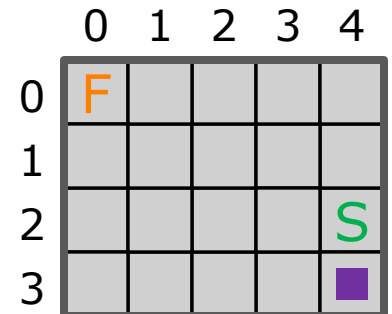
Here is a 4×1 board with *no* holey spider walks.

- `countHSW(4,1, 1,0, 0,0, 3,0)` returns 0.



Here is a 5×4 board. It has 40,887 different holey spider walks.

- `countHSW(5,4, 4,3, 4,2, 0,0)` returns 40887.



Thoughts on Project 4

Writing It [1/8]

We consider how to write `countHSW`.

Requirements in a nutshell:

- Wrapper function `countHSW`.
- Recursive workhorse function `countHSW_recurse`. Prototype this however you want, but the following must be true.
 - It is given a partial solution.
 - It returns number of full solutions based on this partial solution.
 - It does recursive backtracking.
 - It does the bulk of the work.

In the following slides I explain in more detail how I wrote these functions. You may take these as *suggestions* for how you can do it. However, the requirements of the project allow for quite a bit of variation. You do not need to follow my suggestions.

Thoughts on Project 4

Writing It [2/8]

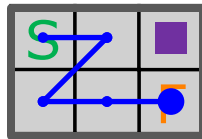
A **partial solution** is a stage on the way to a holey spider walk. Here is the definition of partial solution that I used:

- The walk started on the start square.
- Moves were only made to adjacent squares (N/S/E/W/diagonal) on the board, and not to the hole.
- No square has been visited more than once.
- *However, some squares may not be visited.*

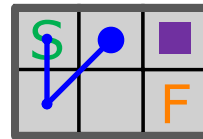
Partial Solutions



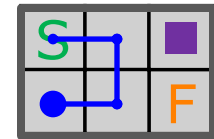
This partial solution is an **empty partial solution**.



This partial solution is a **full solution**.



This partial solution can be extended to a full solution.



This partial solution *cannot* be extended to a full solution.

Thoughts on Project 4

Writing It [3/8]

What data do we need to maintain?

Say a *boardArray* holds an `int` for each square on the board. This is 1 if either the spider has visited the square or it is the **hole**; otherwise it is 0. Thus, 0 means the square needs to be visited.

We also need:

- The board dimensions: x (width) & y (height).
- The **finish** square: x & y .
- The spider's **current position**: x & y .

It would be helpful to know:

- The number of squares left to visit.

Wrap all this
in an object,
if you want.
(I did not.)

Information we do not need to maintain:

Do you see
why not?

- The **start** square.
- The order in which the spider walks through the squares.
- Distinguishing the **hole** from visited squares.

Thoughts on Project 4

Writing It [4/8]

Conceptually, a *boardArray* is a 2-D array.

We can implement it as a vector of vector of `int`.

Construct as follows; each item is initialized to zero:

```
vector<vector<int>> board(dim_x, vector<int>(dim_y, 0));
```

To look up item `i,j`:

```
board[i][j]
```

If we do things as above, then keeping track of `dim_x`, `dim_y` separately is not *required*, but it is convenient to do so.

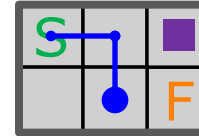
Thoughts on Project 4

Writing It [5/8]

We represent a partial solution with:

- `board: vector<vector<int>>`
- `dim_x, dim_y`: both `int`
- `finish_x, finish_y`: both `int`
- `curr_x, curr_y`: both `int`
- `squaresLeft`: `int`

Partial Solution



Representation

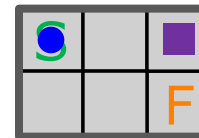
1	1	1
0	1	0

`dim_x/y`: 3, 2
`finish_x/y`: 2, 1
`curr_x/y`: 1, 1
`squaresLeft`: 2

In an **empty partial solution**:

- board items are all 0, except the **start** square and the **hole**.
- `dim_x, dim_y` are the dimensions.
- `finish_x, finish_y` are the coordinates of the **finish** square.
- `curr_x, curr_y` are the coordinates of the **start** square.
- `squaresLeft` is the number of squares on the board minus 2.

Partial Solution



Representation

1	0	1
0	0	0

`dim_x/y`: 3, 2
`finish_x/y`: 2, 1
`curr_x/y`: 0, 0
`squaresLeft`: 4

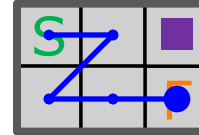
Thoughts on Project 4

Writing It [6/8]

In a **full solution**:

- `dim_x`, `dim_y`, `finish_x`, and `finish_y` are as before.
- board items are all 1.
- `squaresLeft` is 0.
- `curr_x`, `curr_y` are the **finish** square coord's.

Partial Solution



Representation

1	1	1
1	1	1

`dim_x/y:` 3, 2
`finish_x/y:` 2, 1
`curr_x/y:` 2, 1
`squaresLeft:` 0

To test for a full solution, check the last two points above.

```
int countHSW_recurse(vector<vector<int>> board,
    int dim_x, int dim_y, int finish_x, int finish_y,
    int curr_x, int curr_y, int squaresLeft)
{
    if (squaresLeft == 0
        && curr_x == finish_x && curr_y == finish_y)
        return 1; // We have a full solution
    ...
}
```

Thoughts on Project 4

Writing It [7/8]

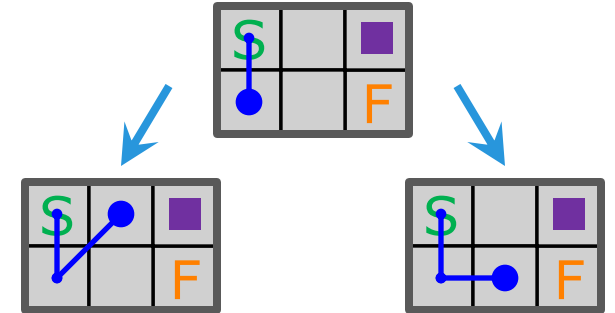
Procedure for workhorse function:

- If we have a full solution, return 1.
- Set *total* to 0.
- For each of the eight squares adjacent to the spider's current position:
 - If this square (1) lies on the board and (2) is not-yet-visited:
 - Move current spider position.
 - Mark new square as visited.
 - Decrement number of squares left.
 - Make recursive call.
 - Add return value to *total*.
 - Restore all changes, except change to *total*.
- Return *total*.

First
check
this.

Then
check
this.

Partial Solutions



Representations

1	0	1
1	0	0

dim_x/y: 3, 2
finish_x/y: 2, 1
curr_x/y: 0, 1
squaresLeft: 3

1	1	1
1	0	0

dim_x/y: 3, 2
finish_x/y: 2, 1
curr_x/y: 0, 1
squaresLeft: 3

1	0	1
1	1	0

dim_x/y: 3, 2
finish_x/y: 2, 1
curr_x/y: 0, 1
squaresLeft: 3

More Suggestions

- Before accessing the data for a square, always check whether the x and y indices of that square are in range!
- If you are careful to leave the board in the same state when `countHSW_recurse` ends as when it began, then you can pass the board by reference, avoiding a time-consuming copy.

```
int countHSW_recurse(vector<vector<int>> & board, ...
```

- There are simple ways to check for some dead ends. Implement one or more and get shorter execution times.

Thoughts on Project 4

Final Notes

Again, you do not have to write your code the way I have outlined. Any code that meets the requirements of the project is acceptable. Once more, in a nutshell:

- Function `countHSW`: prototyped as required.
- Function `countHSW_recurse`: recursive backtracking, takes partial solution and counts full solutions, does the bulk of the work.

But if you have trouble with this kind of thing, then follow my suggestions.

Think first! This project generally requires less writing than other projects, but more thought.

If your code passes all tests, then I will time it, using a special test program that I will not give you. This is *not* for a grade. The names of top performers will be announced in class.

Unit Overview

Algorithmic Efficiency & Sorting

Major Topics

- ✓ ■ Analysis of Algorithms
- ✓ ■ Introduction to Sorting
 - Comparison Sorts I
 - Asymptotic Notation
 - Divide and Conquer
 - Comparison Sorts II
 - The Limits of Sorting
 - Comparison Sorts III
 - Non-Comparison Sorts
 - Sorting in the C++ STL

Review

Part 2

Efficient [noun form: **efficiency**]

- General meaning. Using few **resources**: time, space, etc.
- Specific meaning. Fast—not using much time.
- Other specific meanings when qualified. **Space efficient**, etc.
- Unless we say otherwise, we are talking about the **worst case**—the maximum resource usage for a given input size.

Our *usual* **model of computation**:

- Legal **operations**: no data access except thru provided channels.
- **Basic operations** (the operations we count):
 - A built-in operation on a fundamental type.
 - A call to a client-provided function.
- We are given a list as input. Its **size** is the number of items in it.

It matters
what we count!

Scalable: works well with large problems. (Or, it **scales well**.)

Review

Analysis of Algorithms [2/3]

Algorithm A is **order** $f(n)$ [written $O(f(n))$] if there exist constants k and n_0 such that algorithm A performs no more than $k \times f(n)$ basic operations when given input of size $n \geq n_0$.

We will use big- O every single day for the rest of the semester.

We are interested in the fastest category that an algorithm fits in:

Using Big- O	In Words
$O(1)$	Constant time
$O(\log n)^*$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)^*$	Log-linear time
$O(n^2)$	Quadratic time
$O(c^n)$, for some $c > 1$	Exponential time

Cannot read all of input ↑

.....
Probably not scalable ↓

Faster ↑

Slower ↓

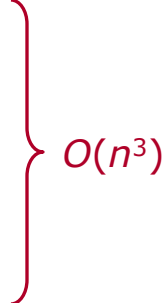
*As we will see, the base of the logarithms does not matter.

I will also allow $O(n^3)$, $O(n^4)$, etc.

When determining big- O , we can collapse any *constant* number of steps into a single step without altering the order.

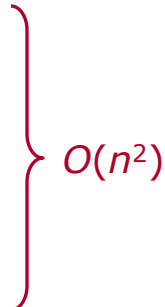
Rule of Thumb. In nested loops, if each is executed n times, or executed i times, where i goes up to n (plus a constant?), then the order is $O(n^t)$ where t is the number of nested loops.

```
for (int i = 0; i < n-4; ++i)
    for (int j = 0; j < i; ++j)
        for (int k = j; k < i+4; ++k)
            ++a[k];
```



$O(n^3)$

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < i-5; ++j)
        for (int k = 0; k < 5; ++k)
            ++arr[j+k];
```



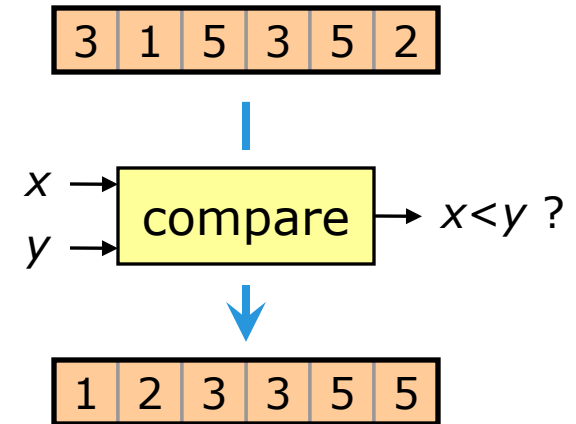
$O(n^2)$

Sort: Place a list in order.

Key: The part of the item we sort by.

Comparison sort: Sorting algorithm that only gets information about item by comparing them in pairs.

A **general-purpose comparison sort** places no restrictions on the size of the list or the values in it.



Analyzing a general-purpose comparison sort:

- (Time) Efficiency
 - Requirements on Data
 - Space Efficiency
 - Stability
 - Performance on Nearly Sorted Data
- In-place** = no large additional space required.
- Stable** = never reverses the relative order of equivalent items.
1. All items close to proper places, OR
2. few items out of order.

Sorting Algorithms Covered

- Quadratic-Time [$O(n^2)$] Comparison Sorts
 - Bubble Sort
 - Insertion Sort
 - Quicksort
- Log-Linear-Time [$O(n \log n)$] Comparison Sorts
 - Merge Sort
 - Heap Sort (mostly later in semester)
 - Introsort
- Special Purpose—Not Comparison Sorts
 - Pigeonhole Sort
 - Radix Sort

Comparison Sorts I

Comparison Sorts I

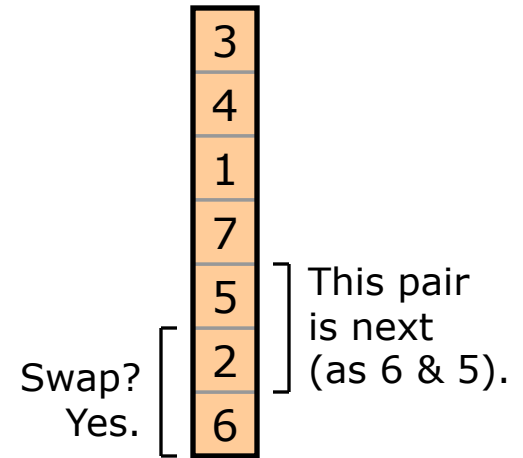
Bubble Sort — Description

We begin with a very simple sort: **Bubble Sort**.

- It is easy to understand and analyze. But we do not use it for anything practical.

Bubble Sort proceeds in a number of **passes**.

- In each pass, we compare consecutive pairs of items. An out-of-order pair is swapped.
- Think of a vertical list, bottom to top. Large items rise like bubbles.
- After the first pass, the last item is the largest.
- So later passes need not go through all the data.



We can improve Bubble Sort's performance on some nearly sorted data—specifically, Type 1 (all items close to proper place):

- In each pass, track whether we have done swaps during that pass.
- If not, then the data were sorted when the pass began. Quit.

Comparison Sorts I

Bubble Sort — CODE

TO DO

- Examine an implementation of Bubble Sort.
- Analyze it.
 - *Coming up.*

See `bubble_sort.cpp`.

Comparison Sorts I

Bubble Sort — Analysis

(Time) Efficiency ☹️

- Bubble Sort is $O(n^2)$.
- Bubble Sort also has an average-case time of $O(n^2)$. ☹️

Requirements on Data 😊

- Bubble Sort does not require random-access data.
- It works on Linked Lists.

Space Efficiency 😊

- Bubble Sort can be done in-place.

Stability 😊

- Bubble Sort is stable.

Performance on Nearly Sorted Data 😊/☹️

- Type 1. An optimized Bubble Sort is $O(n)$ for all items close to their proper spots. 😊
- Type 2. Bubble Sort can be $O(n^2)$ for only one item out of order. ☹️

There are several smileys here, but **these** are more important.

Comparison Sorts I

Bubble Sort — In Practice

Bubble Sort is virtually never used in practice. Its primary purpose is to be an example of an easy-to-understand sorting algorithm.

The other sorts we cover will all be at least a *little bit* practical—and some will be *very* practical.

Comparison Sorts I

Insertion Sort — Description

We can think of Bubble Sort as constructing a sorted sequence in backwards order:

- Find the greatest item (by “bubbling”), then the next greatest, etc.
- So for each **position**, starting with the last, it finds the **item** that belongs there.

Suppose we flip this around.

- Instead of looking through the positions and determining what item belongs in each, look through the given **items**, determine in which **position** each belongs, and then **insert** it in that position.



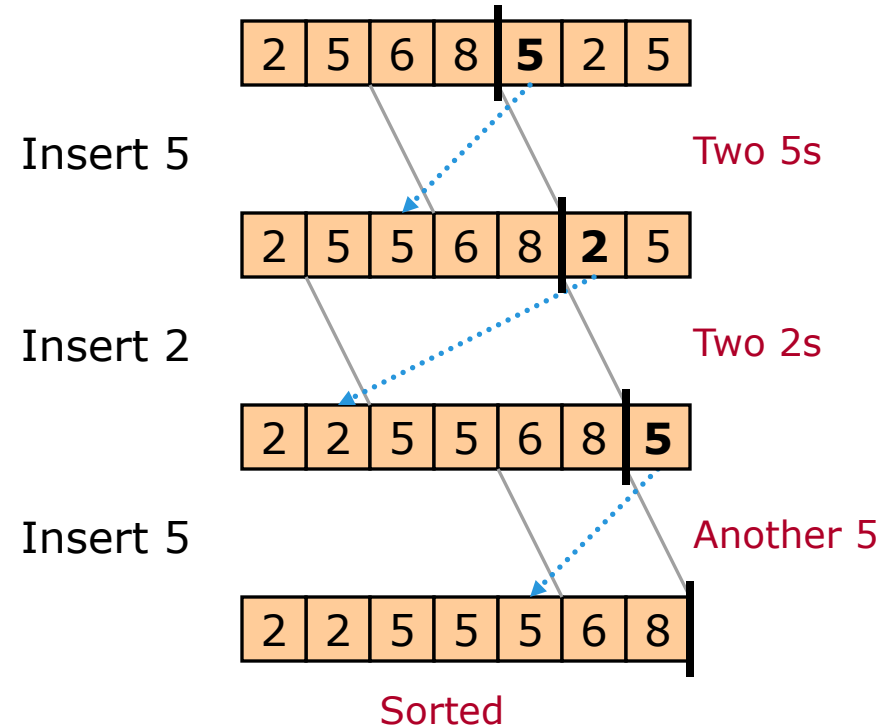
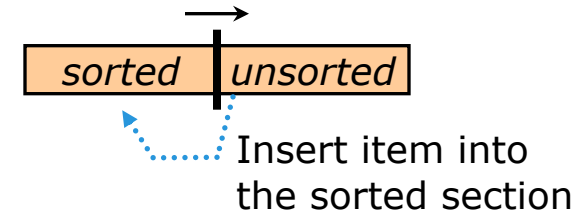
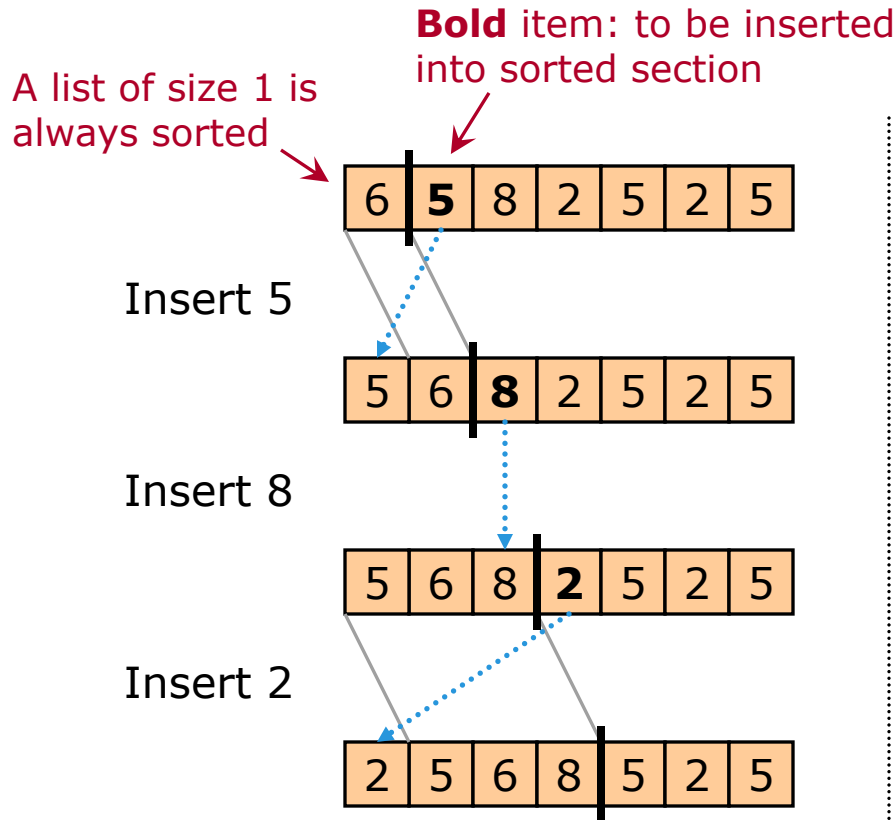
This idea leads to an algorithm called **Insertion Sort**.

- Iterate through the items in the sequence.
- For each, insert it in the proper place among the preceding items.
- Thus, when we are processing item k , we have items $0 \dots k-1$ already in sorted order.

Comparison Sorts I

Insertion Sort — Illustration

Items to the left of the bar are sorted.



Comparison Sorts I

Insertion Sort — How to Search

How do we find the insertion location—that is, the spot in the sorted part of the list where an item should be inserted?

- Sequential Search?
- Binary Search?

We usually use a third option: **backward Sequential Search**—Sequential Search proceeding from back to front.

Why?

- First, Insertion Sort is most useful when the dataset is already nearly sorted. For such data, a backward Sequential Search tends to find the insertion location quickly.
- Second, using Binary Search would not make the algorithm any faster. For an array, we need to go backwards sequentially through the data anyway, since each data item after the insertion location must be moved up. And for a Linked List—or other non-random-access structure—Binary Search is not very fast anyway.

Comparison Sorts I

Insertion Sort — CODE

TO DO

- Implement Insertion Sort.
- Analyze, as before.
 - *Coming up.*

Done. See insertion_sort.cpp.

Comparison Sorts I

Insertion Sort — On `std::move`

`std::move` (<utility>) takes one argument, which it casts to an Rvalue. Use it to force move construction/assignment.

```
a = b;           // Does a copy
a = move(b);     // Does a move
```

`std::move` does not move anything!
It casts to an Rvalue, which makes
its argument *movable*.

The second line of code above is often faster. However, when we do it, we are making an implicit promise: we will not use the current value of `b` again.

```
cout << b;       // BAD!
```

```
b = c;
cout << b;       // Okay
```

There is another `std::move`, in
<algorithm>, taking 3 arguments.
It is the move version of `std::copy`.

Comparison Sorts I

Insertion Sort — Analysis

(Time) Efficiency ☹️

- Insertion Sort is $O(n^2)$.
- Insertion Sort also has an average-case time of $O(n^2)$. ☹️

Requirements on Data 😊

- Insertion Sort does not require random-access data.
- It works on Linked Lists.*

Space Efficiency 😊

- Insertion Sort can be done in-place.

Stability 😊

- Insertion Sort is stable.

Performance on Nearly Sorted Data 😊

- The usual implementation is $O(n)$ for both Type 1* (all items close to proper spots) and Type 2 (few items out of order).

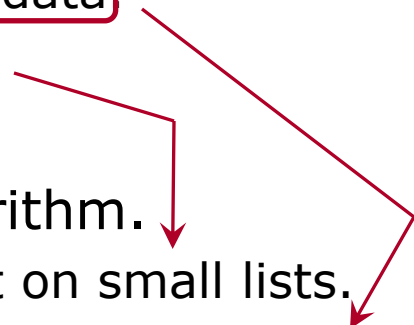
*For one-way sequential-access data, significant extra space usage is required to allow for linear-time sorting of all nearly sorted datasets.

Comparison Sorts I

Insertion Sort — In Practice

Insertion Sort is too slow for general-purpose use.

However, Insertion Sort is useful in certain special cases.

- Insertion Sort is *fast* (linear time) for **nearly sorted data**
 - Insertion Sort is also considered fast for **small lists**.
- 

Insertion Sort often appears as part of another algorithm.

- Optimized sorting code typically does Insertion Sort on small lists.
- Some sorting methods get the data nearly sorted, and then finish with a call to Insertion Sort. *More on this when we cover Quicksort.*