Binary Addition

Decimal / Binary

$$\begin{array}{r} 7 \\ +\ 5 \\ \hline 12 \end{array} \quad \longleftrightarrow \quad \begin{array}{r} 1\ \ 11 \\ 0111 \\ +\ 0101 \\ \hline 1100 \end{array} \quad \longleftarrow \text{Sum}$$

- All 0s in a column ⇒ Sum has a 0 in this column
- One 1 in a column ⇒ Sum has a 1 in this column
- Two 1s in a column ⇒ Sum has a 0 in this column & we Carry a 1 over to the next column
- Three 1s in a column ⇒ Sum has a 1 in this column & we Carry a 1 over to the next column

$a+b$ Pseudocode $(a_n a_{n-1} \dots a_1)_2 + (b_n b_{n-1} \dots b_1)_2$

Carry := 0

for $j := 0$ to $n-1$

  $d := \lfloor (a_j + b_j + \text{Carry})/2 \rfloor$

  $s_j := a_j + b_j + \text{Carry} - 2d$

  Carry := d

$s_n :=$ Carry

Return $(s_n s_{n-1} \dots s_1 s_0)$

Note:

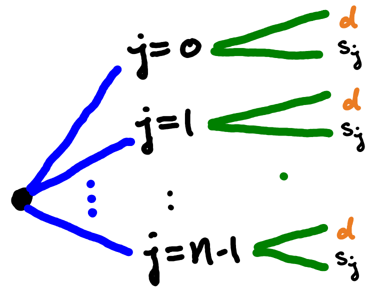
$a_j + b_j + C$		d	s_j
0	⟶	0	0
1	⟶	0	1
2	⟶	1	0
3	⟶	1	1

Compare w/ bullet points above

Elementary Operations

(i) Loop Iterations n

(ii) Addition of bits $2n$



Note: The two different choices of elementary operation above result in "run times" which differ by a constant (in this case it is because 2 additions happen each loop iteration)

Corollary:

The above algorithm for binary addition is $O(n)$

proof: $n \in O(n)$, but also $\underbrace{2n \in O(n)}_{\text{Exercise}}$ $\square$

Binary Multiplication

Multiply by 2

$$\begin{array}{r} 10101 \\ \times \quad 10 \\ \hline 101010 \end{array} \qquad \begin{array}{r} 21 \\ \times 2 \\ \hline 42 \end{array}$$

- Shift all bits to the left & add a 0 to the end
- this idea, together w/ the distributive property is all we need to understand for binary multiplication

$1101 \times 1011 = 1101 \times (1000 + 0010 + 0001)$

$= 1101000 + 11010 + 1101$

From here we use the addition algorithm

$a \cdot b$ **Pseudocode** $(a_n a_{n-1} \ldots a_1)_2 \times (b_n b_{n-1} \ldots b_1)_2$

$O(n)$
```
for j:=0 to n-1
    if b_j = 1 then c_j := a shifted j places
    else c_j := 0
```

recall add(-,-) is $O(n)$

$O(n^2)$
```
product := 0
for j:=0 to n-1
    product := add(product, c_j)
return product
```

Elementary Operations

(i) Loop Iterations $\leq n + n^2$

first loop

second loop calls "add" which has a loop of its own

(ii) Shifts $\leq 1+2+\ldots+(n-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$

Corollary:
The above algorithm for binary multiplication is $O(n^2)$

proof: Both $n+n^2$ & $\frac{n(n+1)}{2}$ are $O(n^2)$

Exercise $\square$

Remark: $\exists$ an algorithm for multiplication that is $O(n^{1.585})$

The Division Algorithm

$a, d \longrightarrow a \text{ div } d,\ a \bmod d$

```
quotient := 0
remainder := a        <- we assume this is positive

while remainder > divisor
    remainder := remainder - divisor
    quotient := quotient + 1
return (quotient, remainder)
```

Since multiplication is repeated addition, division is repeated subtraction

Fact: If the binary expansions of a & d contain fewer than n bits there are $O(n^2)$ algorithms for computing $a \text{ div } d$ & $a \bmod d$

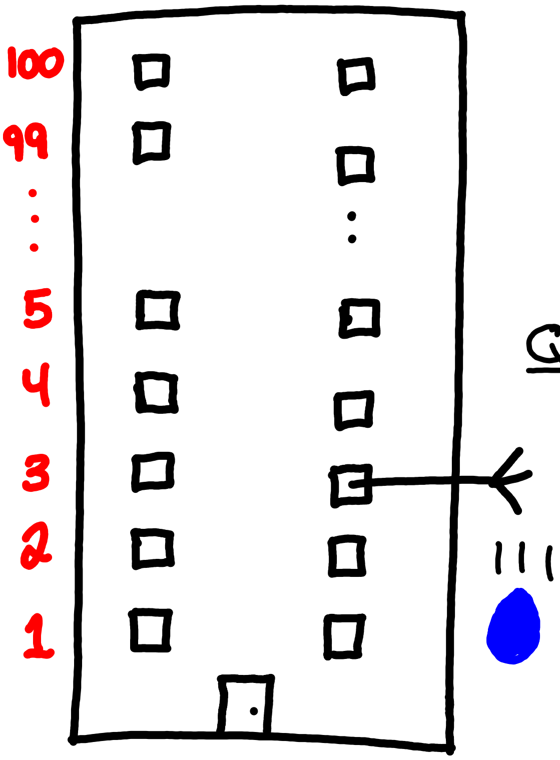
The Euclidean Algorithm

$a, b > 0 \longrightarrow \gcd(a, b)$

```
x := a
y := b
while x ≠ y do
    if x < y then
        y := y - x
    else x := x - y
return x
```

Note: This looks somewhat different from the version of the Euclidean Algorithm we saw before. We will prove that this code always produces the correct output in a little later.

Searching Algorithms



We would like to know the largest floor # from which an egg can be dropped without breaking.

Q: What is the minimum # of egg drops needed to guarantee we have found the correct floor number?

Strategy #1: "Linear Search"

- try dropping an egg out of a first floor window
  - ↳ if it breaks we are done
  - ↳ otherwise move up one floor & repeat

Worst Case we drop 100 eggs in total

But we can do better

Strategy #2: "Binary Search"

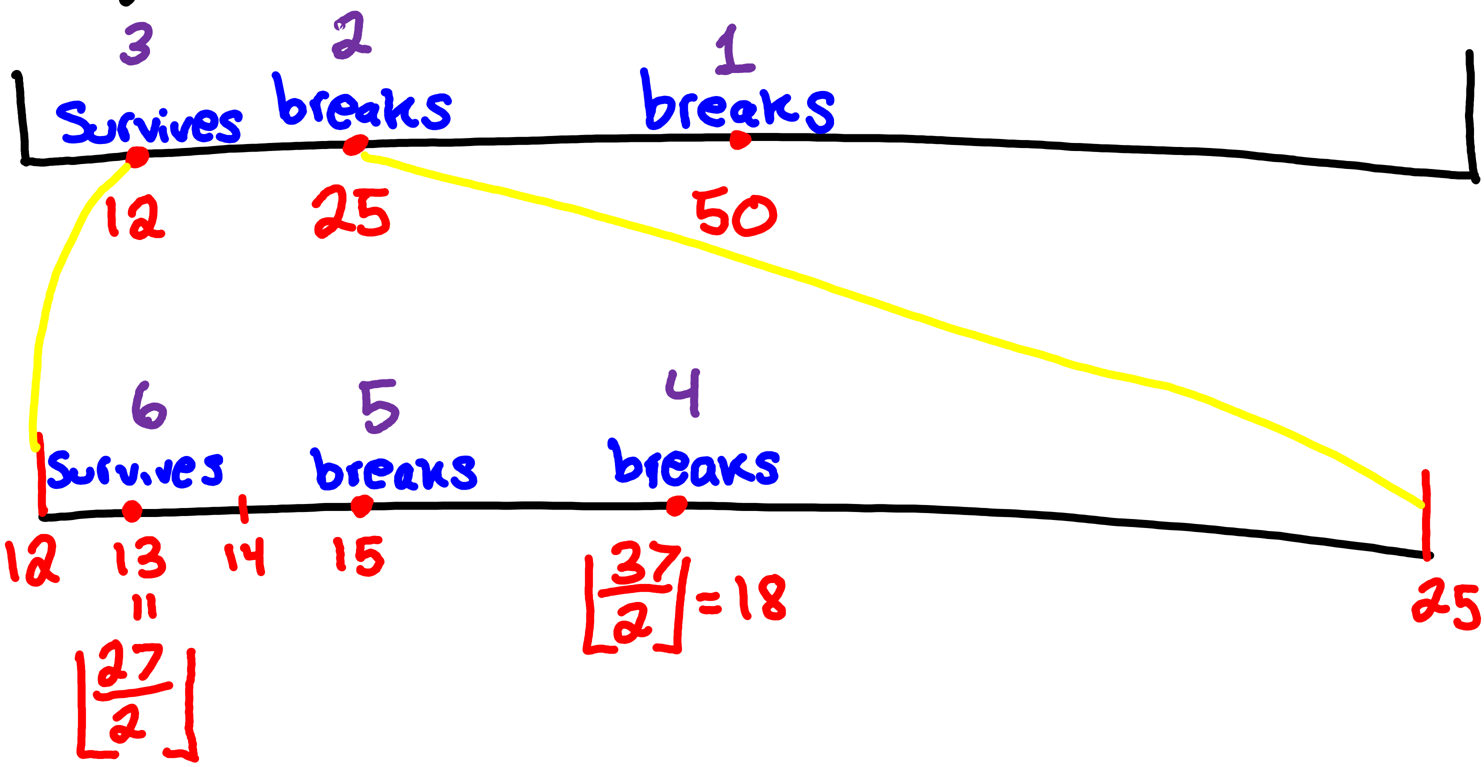
- Drop an egg from floor 50
  - ↳ if it breaks, go to floor 25 & try again
  - ↳ otherwise go to floor 75 & try again

In general, we drop an egg from floor x

↳ if it breaks go to the floor "halfway" (rounding down) between floor x & the highest floor # smaller than x that we have already tested

↳ Otherwise go to the floor "halfway" between floor x & the lowest floor # larger than x that has already been tested

e.g Worst Case



Finally, the last test necessary is floor 14 (the 7th egg drop). If it breaks here we deduce that the answer is floor 13 & otherwise it survives & the answer is floor 14.

$$64 = 2^6 < 100 < 2^7 = 128$$

this method requires $\lceil \log_2(100) \rceil$ egg drops.

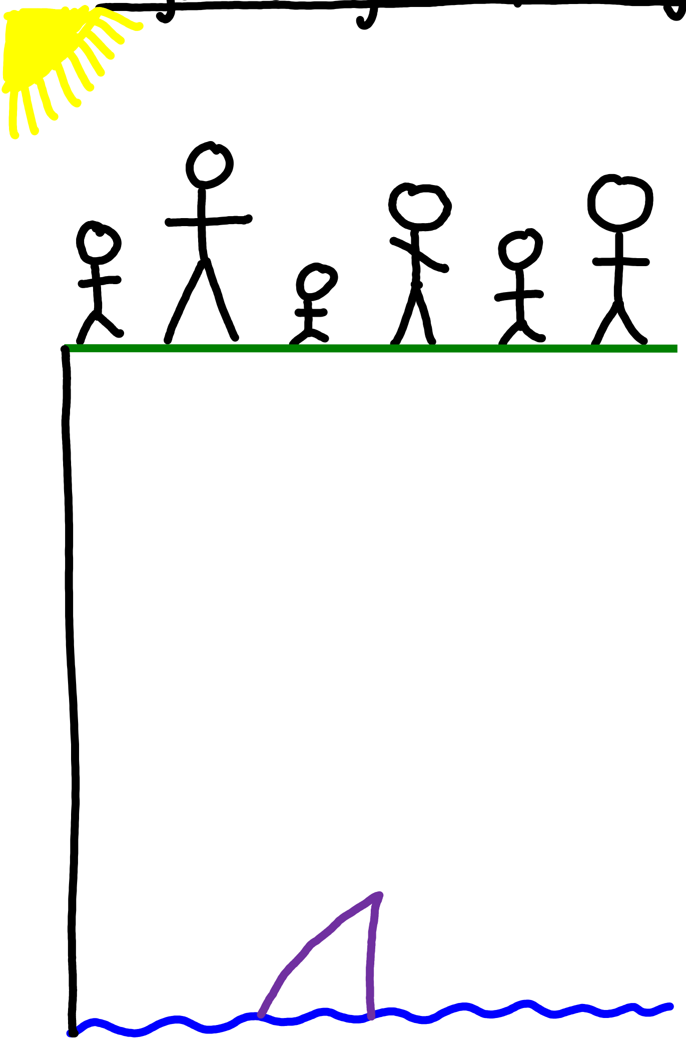
Linear v.s. Binary

Search

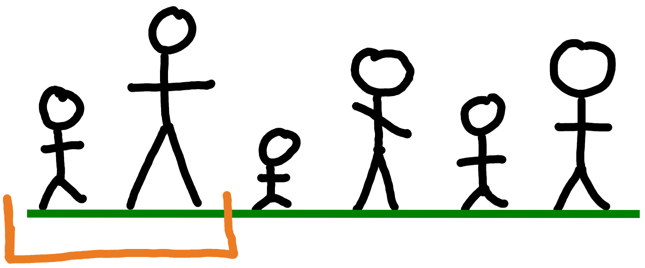
Linear	Binary
$O(n)$	$O(\log(n))$
Always works	requires a sorted list

- When searching an integer array for specific values binary search is faster (because $\log(n) \in O(n)$) but requires a sorted array (i.e. the integers in the array either increase moving left to right or decrease). On the other hand, Linear search is slower (because $\log(n) \notin \Theta(n)$) but does not require a sorted array.

Organizing People by Height at a Great Height

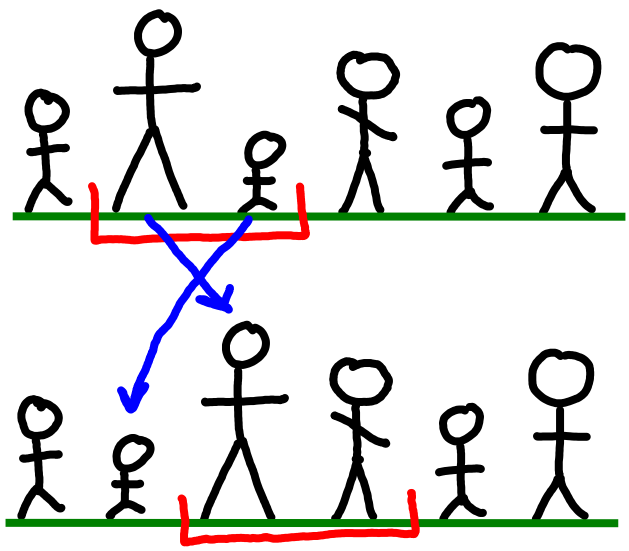


- We're taking family photos on a really thin plank of wood, high above shark infested waters
- Since we would rather not have to repeat this dangerous task, we would like to take at least 1 photo in each possible arrangement
- To change positions on the plank, two adjacent family members must carefully shimmy past one another
- How do we, for example, arrange ourselves by height?

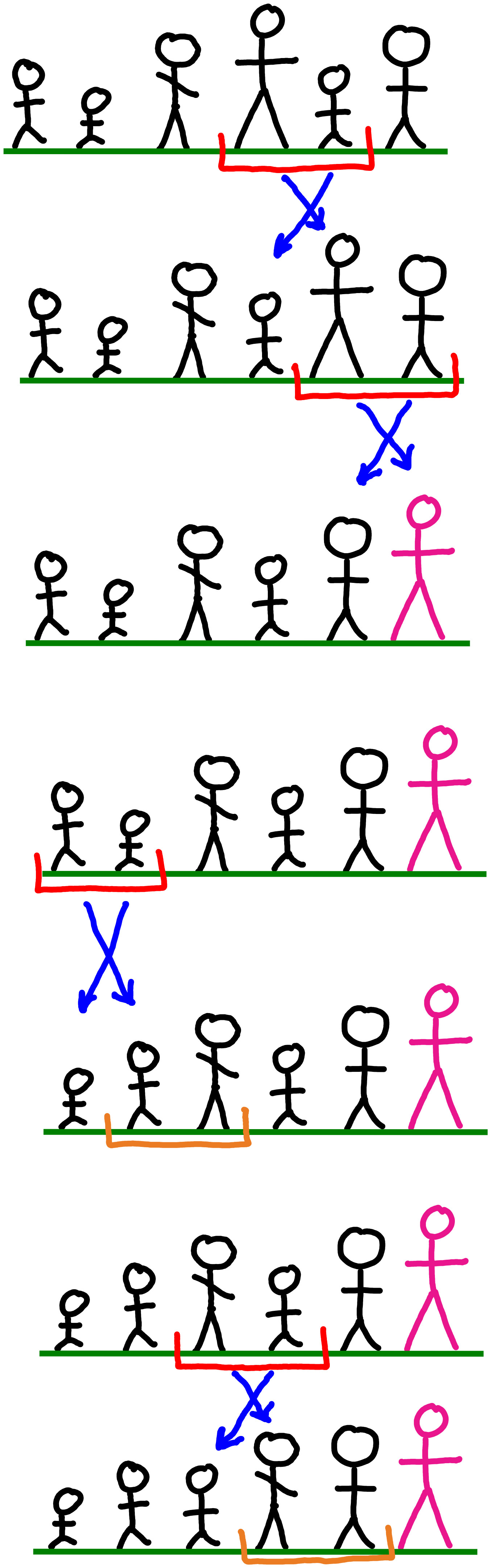


We don't ask these two to move since the one to the left is shorter than the one to the right

moving to the next pair

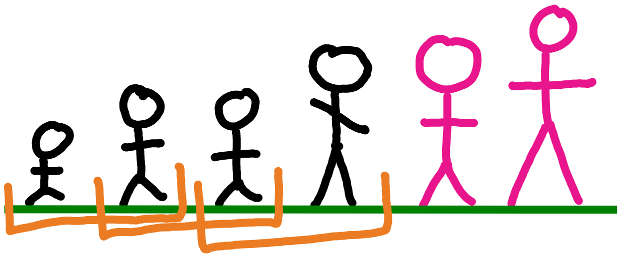


the one on the left is taller than the one on the right, so we ask them to switch their respective positions on the plank

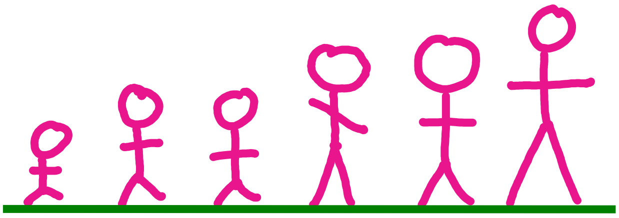


- tallest person is now in the correct spot

- start over from the far left



- Now we are sure that the second tallest is in the correct spot



- Coincidentally, we are Done!

Bubble Sort

Note: After each "pass" from left to right, we place at least one person in their correct position i.e., the tallest remaining unsorted person. So, the sorting happens because the tall people "Bubble" to the top of the array.

pseudocode

$a_1, a_2, \ldots, a_n$ → Sorted by increasing size

for $i := 1$ to $n-1$

  for $j := 1$ to $n-i$

    if $a_j > a_{j+1}$ then swap a_j & a_{j+1}

after i passes through the array the top i largest #s appear in sorted order @ the end of the array. Thus, we can stop comparing them every time

Return $(a_1, a_2, \ldots, a_n)$

i counts the # of times you pass through the array

move from left to right

j tells you your current position in the array. This is where comparisons between elements happen

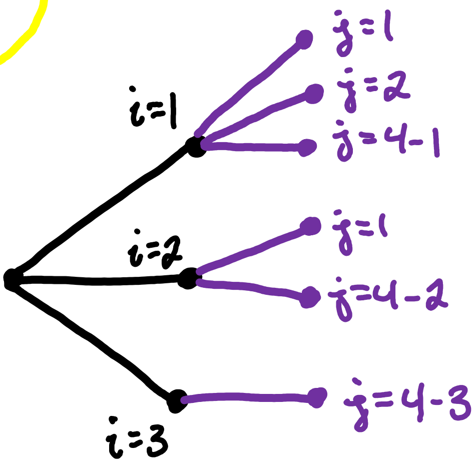
Bubble Sort time Complexity

Elementary operations

(i) Comparisons of integers $\sum_{k=1}^{n-1} k$

e.g. $n=4$

a, b, c, d



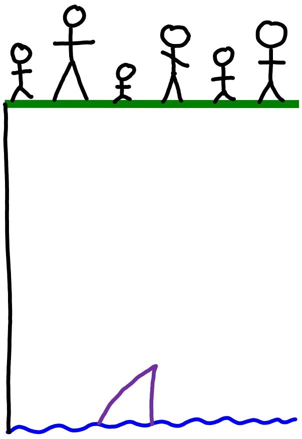
Compare	
a & b	
b & c	3
c & d	
	⊕
a & b	2
b & c	
	⊕
a & b	1

(ii) Entering a loop $\left(\sum_{k=1}^{n-1} k\right) + n-1$

- First loop is entered $n-1$ times
- Each time the second loop is entered whenever a comparison of integers occurs

Recall: $\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$

$\Longrightarrow$ Bubble Sort run time is $O(n^2)$.



How to take every possible family photo

Rather than sort by height, we can sort to an arbitrary final configuration by assigning each person a number. the person we want to be furthest to the right gets the largest number & the person we want standing to the left of them gets the second largest number and so on... so that the person to be positioned all the way to the left has the smallest #. This gives us an unsorted array which we can sort w/ "bubble sort".

Def: An adjacent transposition is a permutation that swaps the position of two adjacent elements & keeps everything else where it was.

e.g.

$123 \xmapsto{\text{ex.}} 132$

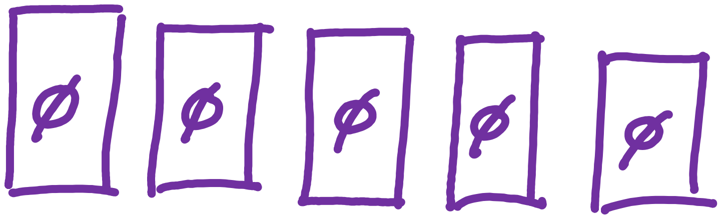
$123 \xmapsto{\text{non-ex.}} 321$

our method of rearranging family members on the plank is by repeated adjacent transpositions

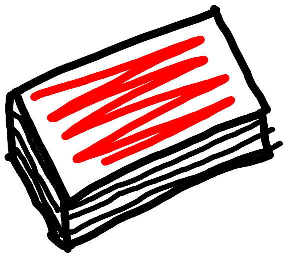
∴ proving the above pseudocode does what we claim it does $\Longrightarrow$ All permutations can be achieved by a sequence of adjacent transpositions. (a well-known fact in math)

Sorting Cards As They Are Dealt

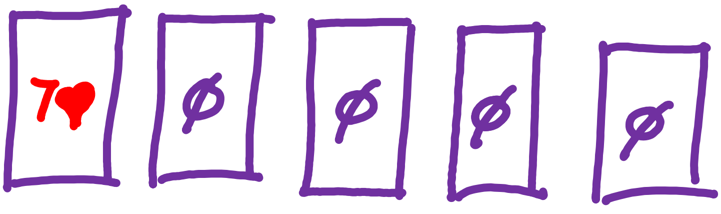
Hand we begin w/ no cards



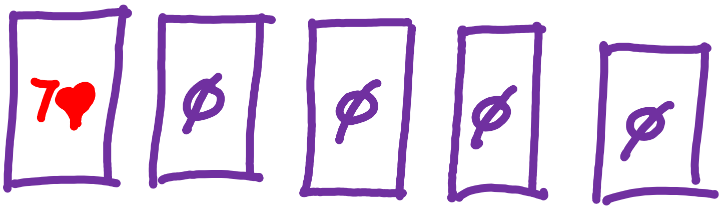
Deck

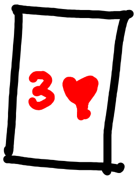


for simplicity, assume we are only dealt hearts

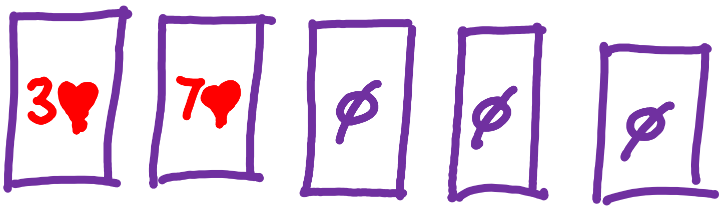


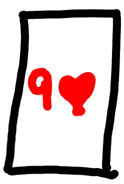
First card we get is the 7 of ♡ there is no preference for its location, so we place it first



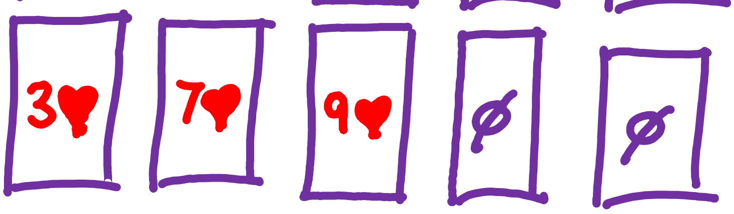


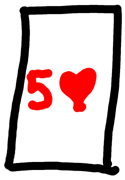
Next we get the 3 of ♡ we place this in our hand so that it is in the correct position relative to the 7 (i.e. before the 7)



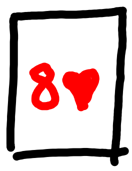


Then we are dealt the 9 of ♡ & we place it after the 7





Next is the 5 of ♡ which we place between the 3 & 7



& Finally, we get dealt the 8 of ♡ which we place between the 7 & 9

this leaves us w/ a sorted hand of cards.

Insertion Sort

Idea: Insert the new card where it belongs with respect to the other cards in your hand

For already filled arrays we treat the leftmost portion as "sorted" & move right placing each # into its correct relative position in the "sorted" piece.

4 1 9 0 7 → 1 4 9 0 7

1 4 9 0 7 → 0 1 4 9 7

1 4 9 0 7 → 0 1 4 7 9

pseudocode

$a_1, a_2, \ldots, a_n$ → Sorted by increasing size

for $j := 2$ to n

$\quad i := 1$

$\quad$ while $a_j > a_i$

$\quad\quad i := i + 1$

locate the correct position of the next element with respect to the sorted part of the array

$\quad m := a_j$

$\quad$ for $k := 0$ to $j - i - 1$

$\quad\quad a_{j-k} := a_{j-k-1}$

$\quad a_i := m$

Place this element in the correct place & adjust other values of the sorted part to accomodate

Return $(a_1, a_2, \ldots, a_n)$

Worst-Case Time Complexity

insertion sort

Elementary operation: Comparisons of integers

integers are only compared in order to enter the while loop

If $j = \alpha$ there are at most $\alpha - 1$ comparisons made by the while loop. j starts @ 2 & goes to n

$$\Rightarrow \le \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$$

comparisons of integers for any array.

$\Rightarrow$ Insertion Sort Run Time is $O(n^2)$

Q: When is this worst-case scenario achieved?

A: When the list is already sorted

Q: When does Bubble Sort require the largest # of adjacent transpositions?

A: When the list begins in reverse sorted order.