Binary Addition

Decimal/Binary

\[
\begin{array}{c}
7 \\
+ \\
5 \\
\hline
12
\end{array}
\quad \leftrightarrow \quad
\begin{array}{c}
111 \\
+ \\
0101 \\
\hline
1100
\end{array}
\]

- All 0s in a column \( \Rightarrow \) Sum has a 0 in this column
- One 1 in a column \( \Rightarrow \) Sum has a 1 in this column
- Two 1s in a column \( \Rightarrow \) Sum has a 0 in this column & we carry a 1 over to the next column
- Three 1s in a column \( \Rightarrow \) Sum has a 1 in this column & we carry a 1 over to the next column

\[a+b\]

Pseudocode

\[(a_n a_{n-1} \ldots a_i)_2 + (b_n b_{n-1} \ldots b_i)_2\]

\[
\text{Carry} := 0
\]

\[
\text{for } j:=0 \text{ to } n-1 \\
\quad d := \lfloor (a_j+b_j+\text{Carry})/2 \rfloor \\
\quad S_j := a_j+b_j+\text{Carry} - 2d \\
\quad \text{Carry} := d \\
\quad S_n := \text{Carry}
\]

Return \((S_n S_{n-1} \ldots S_1 S_0)\)

\[
\begin{array}{cccc}
\text{Note:} & a_j+b_j+C & d & S_j \\
\hline
0 & \rightarrow & 0 & 0 \\
1 & \rightarrow & 0 & 0 \\
2 & \rightarrow & 1 & 1 \\
3 & \rightarrow & 1 & 0 \\
\end{array}
\]

Compose 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 \( 2n \notin O(n) \). \( \square \)

Exercise

Binary Multiplication

Multiply by 2

\[
\begin{array}{c}
10101 \\
\times 10 \\
\hline
101010
\end{array}
\]

\[
21 \times 2 \\
\hline
42
\]

- Shift all bits to the left & add a 0 to the end.

- This idea, together with 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

\[
a \cdot b \quad \text{Pseudocode} \quad (a_n a_{n-1} \ldots a_1)_2 \times (b_n b_{n-1} \ldots b_1)_2
\]

\[ O(n) \quad \begin{cases} 
\text{for } j := 0 \text{ to } n-1 \\
\text{if } b_j = 1 \text{ then } c_j := a \text{ shifted } j \text{ places} \\
\text{else } c_j := 0 \\
\end{cases} \]

\[ \text{product} := 0 \]

\[ O(n^2) \quad \begin{cases} 
\text{for } j := 0 \text{ to } n-1 \\
\text{product} := \text{add}(\text{product}, c_j) \\
\end{cases} \]

\[ \text{return product} \]

**Elementary Operations**

(i) Loop Iterations

\[ \leq n + n^2 \]

(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**

**Remark:** Every algorithm for multiplication that is \( O(n^{1.585}) \)
The Division Algorithm

\[
\begin{align*}
\text{quotient} & := 0 \\
\text{remainder} & := a
\end{align*}
\]

we assume this is positive

\[
\text{while } \text{remainder} > \text{divisor}
\]

\[
\begin{align*}
\text{remainder} & := \text{remainder} - \text{divisor} \\
\text{quotient} & := \text{quotient} + 1
\end{align*}
\]

\[
\text{return } (\text{quotient}, \text{remainder})
\]

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

The Euclidean Algorithm

\[
\begin{align*}
\chi & := a \\
\gamma & := b
\end{align*}
\]

\[
\text{while } \chi \neq \gamma \text{ do}
\]

\[
\begin{align*}
\text{if } \chi < \gamma \text{ then} \\
\gamma & := \gamma - \chi \\
\text{else} & \chi := \chi - \gamma
\end{align*}
\]

\[
\text{return } \chi
\]

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 \( \frac{k}{2} \)

- if it breaks go to the floor "halfway" (rounding down) between floor \( k \)
  & the highest floor \( k \) smaller than \( k \)
  that we have already tested

- otherwise go to the floor "halfway" between floor \( k \) & the lowest floor \( k \)
  larger than \( k \) that has already been tested.

**E.g. Worst Case**

\[ \left\lfloor \frac{37}{2} \right\rfloor = 18 \]
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.

<table>
<thead>
<tr>
<th>Linear</th>
<th>v.s.</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(n))</td>
<td></td>
<td>(O(\log(n)))</td>
</tr>
<tr>
<td>Always works</td>
<td></td>
<td>requires a sorted list</td>
</tr>
</tbody>
</table>

- 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 \(\log(n) \notin O(n)\)) 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**

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

Return (a_1, a_2, ..., a_n)
```
i counts the # of times you pass through the array

\[ \text{move from left to right} \]

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

\[ \sum_{k=1}^{n-1} k \]

eg. \( n=4 \)

\( a, b, c, d \)

- \( i=1 \)
  - \( j=1 \)
    - \( a \& b \)
    - \( 3 \)
  - \( j=2 \)
    - \( b \& c \)
    - \( 3 \)
  - \( j=4-1 \)
    - \( c \& d \)
    - \( 2 \)

\( \Theta \)

- \( i=2 \)
  - \( j=1 \)
    - \( a \& b \)
    - \( 3 \)
  - \( j=4-2 \)
    - \( b \& c \)
    - \( 2 \)
  - \( j=4-3 \)
    - \( a \& b \)
    - \( 1 \)

(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:

\[ \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} \]
⇒ 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 *ex.* 132
123 *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 \( \Rightarrow \) 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

\[ \varnothing \varnothing \varnothing \varnothing \varnothing \]

**Deck**

for simplicity, assume we are only dealt hearts

First Card we get is the 7 of \( \heartsuit \)

there is no preference for its location, so we place it first

Next we get the 3 of \( \heartsuit \)

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 \( \heartsuit \) & 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 left-most portion as "sorted" & move right placing each ♦ into its correct relative position in the "sorted" piece.

\[\begin{align*}
&\downarrow 4 \underline{1} 9 0 7 \\
&\downarrow \underline{14} 9 0 7 \\
&\downarrow 14 9 0 7 \\
&\downarrow 14 9 0 7
\end{align*}\]
Pseudocode

\[
\text{for } j := 2 \text{ to } n \\
i := 1 \\
\text{while } a_j > a_i \\
i := i + 1 \\
m := a_j \\
\text{for } k := 0 \text{ to } j - i - 1 \\
a_j - k := a_j - k - 1 \\
a_i := m \\
\text{Return } (a_1, a_2, ..., a_n)
\]

Worst-Case Time Complexity

Elementary operation: Comparisons of integers

Integers are only compared in order to enter the while loop.

If \( j = n \), there are at most \( n-1 \) comparisons made by the while loop. \( j \) starts at 2 and goes to \( n \).

\[
\Rightarrow \quad \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}
\]

Comparisons of integers for any array.
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 number of adjacent transpositions?
A: When the list begins in reverse sorted order.