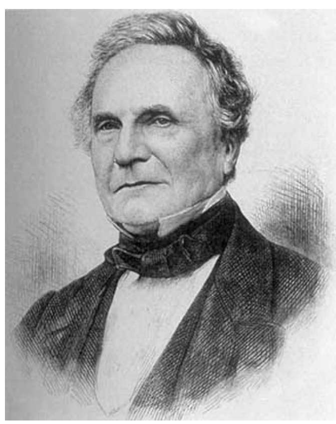
Day 2: Basic of Algorithms Analysis

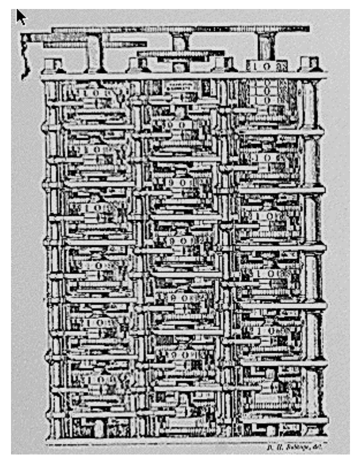
"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - Francis Sullivan

Computational Tractability

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - Charles Babbage



Charles Babbage (1864)



Analytic Engine (schematic)

Computational Tractability

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N, and see how this scales with N.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Desirable scaling property. When the input size increases by a factor of 2, the algorithm should only slow down by some constant factor C.

> There exists constants c > 0 and d > 0 such that on every input of size N, its running time is bounded by $c N^d$ steps.

Def. An algorithm is efficient if it has polynomial running time.

Justification. It really works in practice!

Why It Matters

Run time in nanoseconds -->		$1.3 N^3$	$10 N^2$	$47 N \log_2 N$	$48 N$
Time to solve a problem of size	1000	1.3 seconds	10 msec	0.4 msec	0.048 msec
	10,000	22 minutes	1 second	6 msec	0.48 msec
	100,000	15 days	1.7 minutes	78 msec	4.8 msec
	million	41 years	2.8 hours	0.94 seconds	48 msec
	10 million	41 millennia	1.7 weeks	11 seconds	0.48 seconds
Max size problem solved in one	second	920	10,000	1 million	21 million
	minute	3,600	77,000	49 million	1.3 billion
	hour	14,000	600,000	2.4 trillion	76 trillion
	day	41,000	2.9 million	50 trillion	1,800 trillion
N multiplied by 10, time multiplied by		1,000	100	10+	10

Reference: More Programming Pearls by Jon Bentley

Orders of Magnitude

Seconds	Equivalent
1	1 second
10	10 seconds
10^2	1.7 minutes
10^3	17 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
. . .	forever
10^{17}	age of universe

Meters Per Second	Imperial Units	Example
10^{-10}	1.2 in / decade	Continental drift
10^{-8}	1 ft / year	Hair growing
10^{-6}	3.4 in / day	Glacier
10^{-4}	1.2 ft / hour	Gastro-intestinal tract
10^{-2}	2 ft / minute	Ant
1	2.2 mi / hour	Human walk
10^2	220 mi / hour	Propeller airplane
10^4	370 mi / min	Space shuttle
10^6	620 mi / sec	Earth in galactic orbit
10^8	62,000 mi / sec	1/3 speed of light

Powers of 2		
	2^{10}	thousand
	2^{20}	million
	2^{30}	billion

Reference: More Programming Pearls by Jon Bentley

Asymptotic Order of Growth

Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$, .

Slight abuse of notation. $T(n) = O(f(n))$.

Vacuous statement. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.

Properties

Transitivity. If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.

Additivity. If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.

Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants a, b > 0.

↑ can avoid specifying the base, assuming it is a constant

Logarithms. For every x > 0, $\log n = O(n^x)$.

↑ log grows slower than every polynomial

Exponentials. For every r > 1 and every d > 0, $n^d = O(r^n)$.

↑ every exponential grows faster than every polynomial

Linear Time: $O(n)$

Linear time. Running time is at most a constant factor times the size of the input.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a_1
for i = 2 to n {
   if (a_i > max)
      max ← a_i
}
```

Linearithmic Time: O(n log n)

Linearathmic time. Arises in divide-and-conquer algorithms.

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane (x_1, y_1), …, (x_n, y_n), find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x1 - x2)^2 + (y1 - y2)^2
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)^2 + (yi - yj)^2      ← don't need to take square roots
      if (d < min)
         min ← d
   }
}
```

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. ← Chapter 5

Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```
foreach set Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
   }
}
```

Polynomial Time: $O(n^k)$ Time

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

$O(n^k)$ solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set = $O(k^2)$.
- Number of k element subsets = $\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \le \frac{n^k}{k!}$
- $O(k^2\, n^k / k!) = O(n^k)$.

assuming k is a constant

Exponential Time

Independent set. Given a graph, what is maximum size of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* ← φ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

Two Key Types of Algorithms

Iterative Algorithms
Recursive Algorithms

Algorithms:

- Precondition(s): what is true when the algorithm starts
  - If it isn't true, the algorithm can do whatever it wants
  - Preconditions are statements about the input
- Postconditions: What is guaranteed to be true after the algorithm completes
  - Postconditions are statements about the output

Iterative Algorithms

Take one step at a time towards the final destination

```
loop (until done)
    take step
end loop
```

Loop Invariants

A good way to structure iterative algorithms

- Store the key information you currently know in some data structure
- In the loop
  - take a step towards destination
  - by making a simple change to the data

Maintain Loop Invariant

If algorithm is in a safe place, it doesn't move to an unsafe location

- That is, loop invariant is always maintained

But, how do we know it starts in a safe place?

- We must establish the loop invariant based on the preconditions of the algorithm

Loop invariant is always true

That's why it's called invariant

How do we know it is always true

- It starts true
- Each time through the loop it stays true
- Induction tells us it is always true

Ending the Algorithm

Define an exit condition

Termination

- With sufficient progress, the exit condition will be met

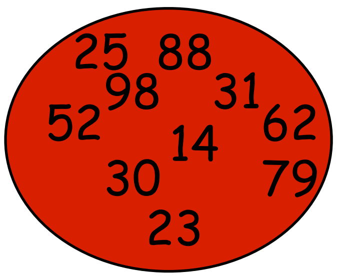
When we exit

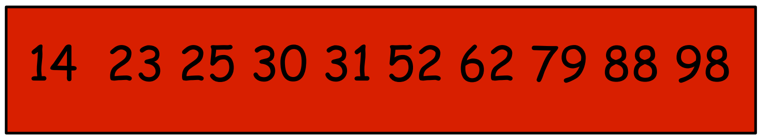
- We know the exit condition is true
- We know the loop invariant is true
- From these, we must establish the postcondition is true

Insertion Sort Example

Precondition: input is list of (possibly repeated) numbers

Postcondition: Output is list of same numbers in non-decreasing order





Define Loop Invariant

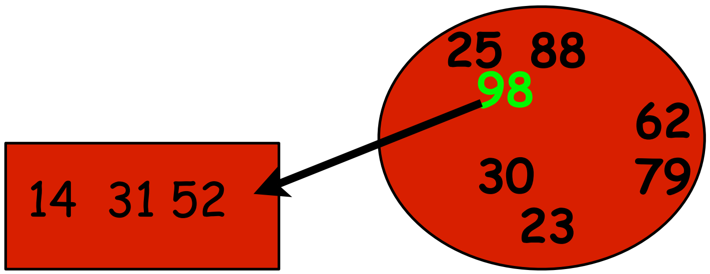
Some subset of the elements are sorted

Remaining elements are off to the side

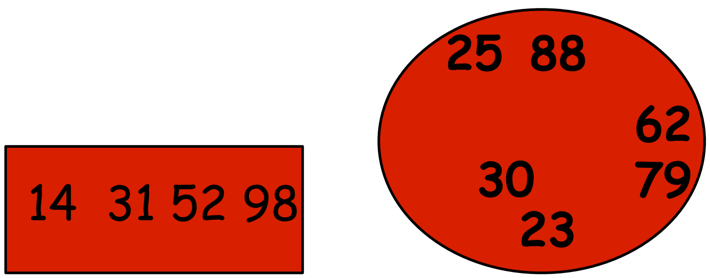
Loop

Choose an element from unsorted elements

Insert it where it belongs elements



before loop

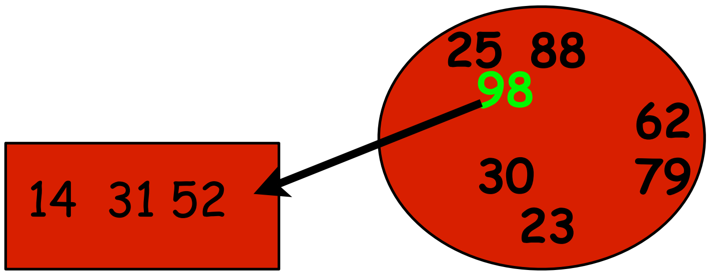


after loop

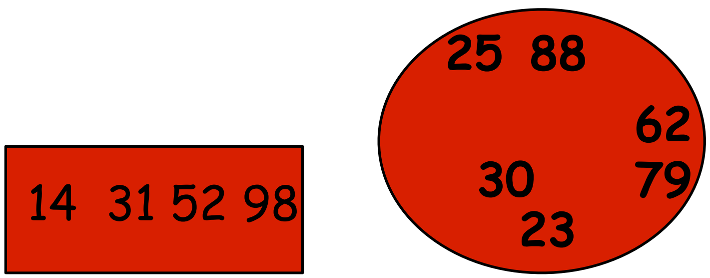
Making progress

Progress

- Amount of the input consumed

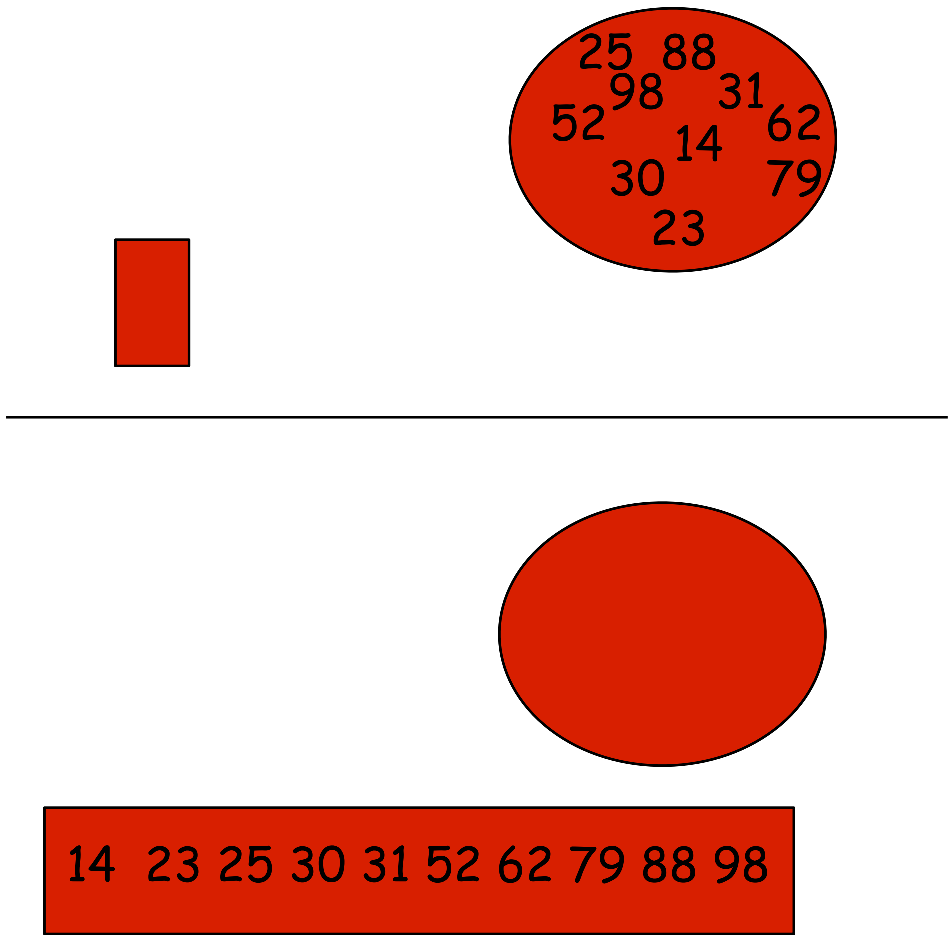


7 to go



6 to go

Beginning and Ending



n to *go*

0 to *go*

Pseudocode

InsertionSort(B)

bag=set with duplicates

Precondition: B is bag of elements

Postcondition: returns L, list of each element from B in non-decreasing order

```
L = empty list
R = B
loop
   loop-invariant: L is a non-decreasing list of elements where R U L = B
   exit when R is empty
   Remove an element e from R
   Insert e into the appropriate location in L
end loop
return L
```

Loop Invariant Example

Initialization

- Since L is empty, L is non-decreasing list of elements
- Since R=B, R U L = B

Maintenance

- L'' is L' with an element added to it in the appropriate location. Therefore, L'' is a non-decreasing list of elements
- B = R' U L' = (R' - e) U (L' U e) = R'' U L''

Termination

- R is empty (by exit condition)
- R U L = B (by LI)
- {} U L = B -> L = B
- L is a non-decreasing list of elements (by LI)

For Tuesday (Day 3)

Read Chapter 4 of Kleinberg