From *Designing a Digital Future*

Progress in algorithms beats Moore's law

Everyone knows Moore's Law — a prediction made in 1965 by Intel co-founder Gordon Moore that the density of transistors in integrated circuits would continue to double every 1 to 2 years.

Even more remarkable — and even less widely understood — is that in many areas, *performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed.*

In the field of numerical algorithms, the improvement can be quantified. Here is just one example. A benchmark production planning model solved using linear programming would have taken 82 years to solve in 1988, using the computers and the linear programming algorithms of the day. Fifteen years later — in 2003 — this same model could be solved in roughly 1 minute, an improvement by a factor of roughly 43 million. Of this, a factor of roughly 1,000 was due to increased processor speed, whereas a factor of roughly 43,000 was due to improvements in algorithms!

Why study algorithms?

- It's all about efficiency!
- We will make heavy use of *abstractions*.
- Solving difficult problems, solving them fast, and figuring out when problems simply cannot be solved fast.

Definition of a Problem

A *problem* is a collection of input-output pairs that specifies the desired behavior of an algorithm.

Example (Sorting Problem)

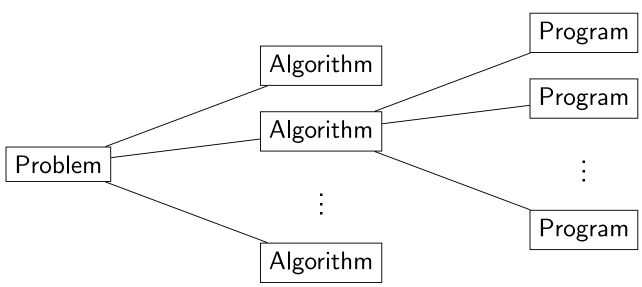
- `[20, 3, 14, 7]`, `[3, 7, 14, 20]`
- `[13, 18]`, `[13, 18]`
- `[5, 4, 3, 2, 1]`, `[1, 2, 3, 4, 5]`
- $\vdots$

Algorithm Definition

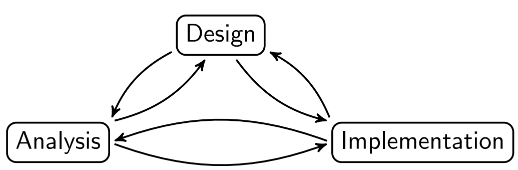
An *algorithm* is a specific way to actually compute the function defined by some problem.

- Must produce correct output for every valid input
- Must terminate in a finite number of steps
- Behavior is *undefined* on invalid input
- Independent of any programming language or architecture

One-to-many relationships



Components of Algorithms



Foci of the course

- **Design**: How to come up with efficient algorithms for all sorts of problems
- **Analysis**: What it means for an algorithm to be "efficient", and how to compare two different algorithms for the same problem.
- **Implementation**: Faithfully translating a given algorithm to an actual, usable, fast program.

Sorted Array Search Problem

Problem: Sorted array search
Input:

- A, sorted array of integers
- x, number to search for

Output:

- An index k such that $A[k] = x$, or `NOT_FOUND`

Algorithm: `linearSearch`
Input: (A, x), an instance of the *Sorted Array Search* problem

```
i = 0
while i < length(A) and A[i] < x do
    i = i + 1
if i < length(A) and A[i] = x then return i
else return NOT_FOUND
```

Algorithm: binarySearch
Input: (A, x), an instance of the *Sorted Array Search* problem

```
left = 0
right = length(A)-1
while left < right do
    middle = floor( (left+right)/2 )
    if x <= A[middle] then
        right = middle
    else if x > A[middle] then
        left = middle+1
    end if
end while
if A[left] = x then return left
else return NOT_FOUND
```


Algorithm: gallopSearch
Input: (A, x), an instance of the *Sorted Array Search* problem

```
i = 1
while i < length(A) and A[i] <= x do
    i = i * 2
left = floor(i/2)
right = min(i, length(A)) - 1
return binarySearch(A[left..right])
```


Loop Invariants

1. **Initialization**: The invariant is true at the beginning of the first time through the loop.
2. **Maintenance**: If the invariant is true at the beginning of one iteration, it's also true at the beginning of the next iteration.
3. **Termination**: After the loop exits, the invariant PLUS the loop termination condition tells us something useful.

Choices in Implementation

- What programming language to use
- What precise language constructs to use (For example, should the list be an array or a linked list? Should we actually call the "length" function on the list every time, or save it in a variable?)
- What compiler to use, and what compiler options to compile with.
- What machine/architecture to run on

Timing Experiments

Input	x	Result	linear	binary	gallop
`[6 7 8]`	4	`NOT`	5	5	7
`[27 50 62 78 ... 180]`	62	2	6	7	12
`[3 6 23 27 ... 990]`	500	`NOT`	76	14	25
`[7 11 14 17 ... 99997]`	19	4	8	31	15
`[14 17 28 58 ... 999992]`	966	99	128	53	27
`[0 2 2 3 ... 9998]`	9999	`NOT`	12108	35	59

- Which one is the fastest?

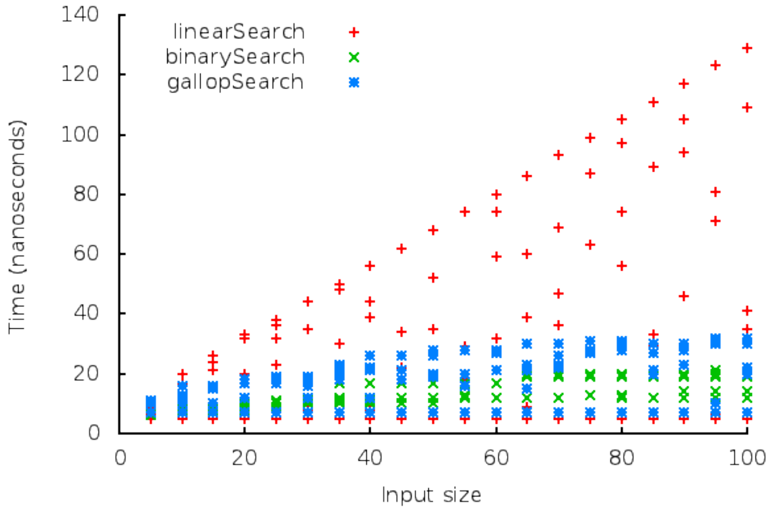
Measure of Difficulty

- Need a way to put timings in context — should spend more time on *harder* inputs.
- Need to *sort* the data so we can make sense of it.

Solution: assign a *difficulty measure* to each input.

Most common measure: **input size**, n.

Search times plot

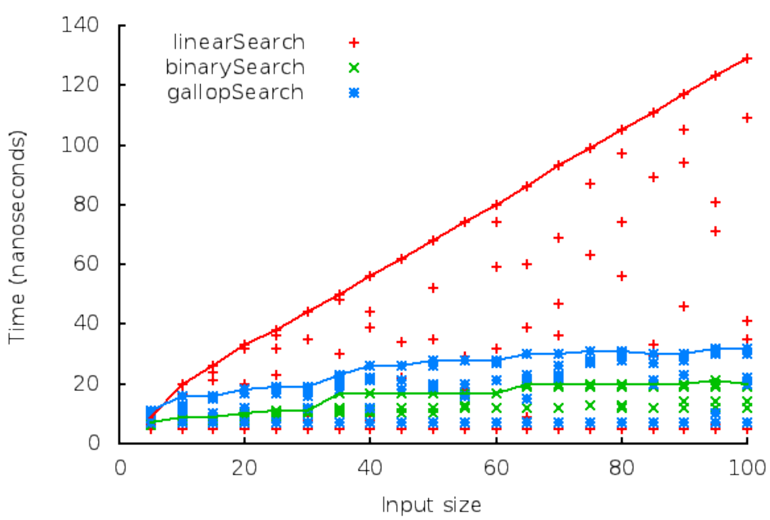


Making a single function for run-time

- **Best-case**: Choose the best (smallest) time for each size
- **Worst-case**: Choose the worst (largest) time for each size
- **Average-case**: Choose the average of all the timings for each size

Of these, **the worst-case time is the usually the most significant**.

Worst-case of search algorithms



Shortcomings of experimental comparison

- It depends on the machine.
- It depends on the implementation.
- It depends on the examples chosen for each size.
- It depends on the sizes chosen.
- Can't describe *how much better* one algorithm is than another.
- Implementations are expensive (time, cost) to create.

Formal analysis will overcome these shortcomings, but requires some more simplifications.

Abstract Machine

To achieve *machine independence*, we usually count the number of operations in an *abstract machine model* such as a RAM.

That's too hardcore for us. Instead, we will count:

Definition (Primitive Operation)

A *primitive operation* is one that can be performed in a fixed number of steps on any modern architecture.

- Intentionally vague definition
- Examples: integer addition, memory lookup, comparison

Primitive count analysis

Asymptotic Notation

- Counting primitive operations exactly is **too precise** and doesn't help to **compare algorithms**
- Solution: Big-O, Big-Ω, Big-Θ

Definition (Big-O Notation)

Given two functions $T(n)$ and $f(n)$, that always return positive numbers, $T(n) \in O(f(n))$ if and only if there exist constants $c, n_0 > 0$ such that, for all $n \geq n_0$, $T(n) \leq cf(n)$.

Big-O Simplification Rules 1

Constant multiple rule

If $T(n) \in O(f(n))$ and $c > 0$, then $T(n) \in O(c * g(n))$.

Domination rule

If $T(n) \in O(f(n) + g(n))$, and $f(n) \in O(g(n))$, then $T(n) \in O(g(n))$.
(In this case, we usually say that g "dominates" f.

Transitivity rule

If $T(n) \in O(f(n))$ and $f(n) \in O(g(n))$, then $T(n) \in O(g(n))$.

Big-O Simplification Rules 2

Addition rule

If $T_1(n) \in O(f(n))$ and $T_2(n) \in O(g(n))$, then
$T_1(n) + T_2(n) \in O(f(n) + g(n))$.

Multiplication rule

If $T_1(n) \in O(f(n))$ and $T_2(n) \in O(g(n))$, then
$T_1(n) * T_2(n) \in O(f(n) * g(n))$.

Trivial rules

For any positive-valued function f:

- $1 \in O(f(n))$
- $f(n) \in O(f(n))$

Big-Ω and Big-Θ

Definition (Big-Ω)

$T(n) \in \Omega(f(n))$ if and only if $f(n) \in O(T(n))$.

Definition (Big-Θ)

$T_1(n) \in \Theta(T_2(n))$ if and only if both $T_1(n) \in O(T_2(n))$ and $T_2(n) \in O(T_1(n))$.

- Which of the previous rules apply for these?

Worst-case running times

- `linearSearch` is $\Theta(n)$ in the worst case
- `binarySearch` is $\Theta(\log n)$ in the worst case
- `gallopSearch` is $\Theta(\log n)$ in the worst case too!

- What does this all mean?

Don't mix up worst/best/average case
with big-O/big-Ω/big-Θ.

Different difficulty measure

- Observation: `linearSearch` and `gallopSearch` perform better when the search key x is very small.
- Alternate difficulty measure: m, the least index such that $A[m] \geq x$.
- Re-do the analysis in terms of m and n.

A different cost function

What if we counted *comparisons* instead of primitive operations?

- `linearSearch`:
- `binarySearch`:
- `gallopSearch`:

Conclusions

- **Which search algorithm is the best?**
- Design, Analysis, Implementation
- Problem, Algorithm, Program
- Best-case, worst-case, and average-case
- Big-O, Big-Ω, Big-Θ