# crash course in theoretical computer science

Glencora Borradaile
Oregon State University
eecs.orst.edu/~glencora/other/tcscrashcourse.pdf

theoretical computer science
= complexity (What are the limits of computation?)
+ algorithms (Design within those limits?)

[follow the links to learn more]

## what is computation?

- solving problems with a (restricted) set of operations
- a better name for *computer science*

## abstract model of computation: the Turing machine

a tape *(memory)*
at any moment *reads* one scanned symbol *(bus)*
can alter scanned symbol according to a finite set of elementary operations *(register)*

(remains a good model for modern computers)

## what is computable? what is incomputable?

- product of two integers is computable
- Entscheidungsproblem is incomputable

## of the computable, what is efficiently computable?

# larger problems = longer computation

eg. computing $761498762598 \times 319870897543$ takes longer than computing $32 \times 54$

$T(n, X, A)$ = **<u>time</u> to solve instance of <u>size</u> $n$ of problem $X$ using algorithm $A$**

= # <u>computational steps</u> = # bits to represent instance

= Turing machine operations

e.g. what is $T(2n, \text{product of two } n \text{ bit numbers}, \text{grade-school})$?

at most $n$ bit multiplications + $n$ bit additions (for the *carry*) per *row*

at most $n$ bit additions per *column*

at most $2n$ columns and $n$ rows

or $4n^2$ bit additions/multiplications

or at most $k(4n^2)$ Turing machine steps for some constant $k$

$O(n^2)$ computational steps

$O(n^2)$ time on *any* single processor

**algorithm analysis:** for a particular $X$ and $A$, what is $T(n, X, A)$?

**algorithm design:** for a particular $X$, find $A$ to minimize $T(n, X, A)$ for all $n$

# efficiently means quickly

## when is $A$ efficient? what values of $T(n, X, A)$ are good?

$$\text{faster} \quad \underbrace{O(n) \quad O(n^2) \quad O(n^3) \quad O(n^{10})}_{\text{polynomial}} \quad n^{\log n} \quad \underbrace{O(2^n) \quad O(n!) \quad O(n^n)}_{\text{exponential}} \quad \text{slower}$$

## polynomial ≈ practical

if $T(n, X, A)$ is $O(n^c)$

- in twice the time, can solve problems $2^{1/c}$ *times* bigger
- if a processor gets twice as fast, can solve problems $2^{1/c}$ *times* bigger in the same time

## exponential ≈ impractical

if $T(n, X, A)$ is $O(c^n)$

- in twice the time, can solve problems bigger by $\log_c 2$ *additively*
- if a processor gets twice as fast, can solve problems bigger by $\log_c 2$ *additively*

## million-dollar question: P v NP

P = set of (decision) problems that can be solved in polynomial time
(on a deterministic Turing machine)
e.g. is this number divisible by this other number?

NP = set of (decision) problems that can be solved in polynomial time
(on a *non*-deterministic Turing machine)
e.g. is this boolean formula satisfiable?

NP = set of (decision) problems with 'yes' answers verifiable in polynomial time
(on a deterministic Turing machine)

*co-NP = set of (decision) problems with 'no' answers verifiable in polynomial time*
*(on a deterministic Turing machine)*
*e.g. is this boolean formula a tautology?*

[Venn diagram of P, NP, co-NP]

# a direction for showing P = NP

design a poly-time algorithm for every problem in NP
what are all the problems in NP? this could take a long time
start with the most computationally-difficult problem

### hard problems

problem $X$ is NP-hard $\iff$
poly-time algorithm for $X \implies$ poly-time algorithm $\forall Y \in$ NP
($\implies$ P = NP)

**Cook-Levin Theorem** boolean formula satisfiability is NP-hard

more generally:

*problem $X$ is $C$-hard $\iff$*
*poly-time algorithm for $X \implies$ poly-time algorithm $\forall Y \in C$*

[Venn diagram of P, NP, NP-hard]

## reductions

problem $X$ *reduces* to problem $Y$
    if algorithm for $X$ can be designed using algorithm for $Y$

problem $X$ *poly-time* reduces to problem $Y$
    if a *poly-time* algorithm for $X$ can be designed using a *poly-time* algorithm for $Y$

## more definitions of hardness

problem $X$ is NP-hard $\iff$ every problem in NP can be poly-time reduced to $X$
problem $X$ is NP-hard $\iff$ a known NP-problem can be poly-time reduced to $X$

e.g. *boolean-formula satisfiability* reduces to *graph Hamiltonicity*
    so, *graph Hamiltonicity* $\in$ NP-hard

### take-home lesson

if you can show your problem is NP-hard (by reducing a known NP-hard problem to it), then you shouldn't look for a poly-time algorithm to solve your problem

# designing poly-time algorithms

## example problem: max subarray

given array of small integers $a[1, \ldots, n]$, compute

$$\max_{i \le j} \sum_{k=i}^{j} a[k]$$

e.g. $\textsc{MaxSubarray}([31, -41, \mathbf{59}, \mathbf{26}, -\mathbf{53}, \mathbf{58}, \mathbf{97}, -93, -23, 84]) = 187$

## algorithmic design techniques

1. enumeration
2. iteration
3. simplification & delegation *(aka divide & conquer)*
4. recursion inversion *(aka dynamic programming)*

## enumeration for max subarray

evaluate every possible solution

```
MaxSubarray(a[1,...,n])
    for each pair (i,j) with 1 ≤ i < j ≤ n
        compute a[i]+a[i+1]+···+a[j-1]+a[j]
        keep max sum found so far
    return max sum found
```

**analysis** $(O(n^2)$ pairs$) \times (O(n)$ time to compute each sum$) = O(n^3)$ time

# iteration for max subarray

don't compute sums from scratch:

$\sum_{k=i}^{j} a[k]$ can be computed from $\sum_{k=i}^{j-1} a[k]$ in $O(1)$ time

(really just clever enumeration)

```
MaxSubarray(a[1,...,n])
    for i = 1, ..., n
        sum = 0
        for j = i, ..., n
            sum = sum + a[j]
            keep max sum found so far
    return max sum found
```

**analysis** $(O(n)\ i\text{-iterations}) \times (O(n)\ j\text{-iterations}) \times (O(1) \text{ time to update sum}) = O(n^2)$

## simplification & delegation for max subarray

max subarray either has value

- $\textsc{MaxSubarray}(a[1, \ldots, \frac{n}{2}])$,
- or $\textsc{MaxSubarray}(a[\frac{n}{2}, \ldots, n])$,
- or $\textsc{MaxSuffix}(a[1, \ldots, \frac{n}{2}])$+$\textsc{MaxPrefix}(a[\frac{n}{2}, \ldots, n])$

compute $\textsc{MaxSuffix}$ and $\textsc{MaxPrefix}$ in linear time by modifying previous algorithm

**divide & conquer**

$$\textsc{MaxSubarray}(a[1, \ldots, n]) = \max \begin{cases} \textsc{MaxSubarray}(a[1, \ldots, \frac{n}{2}]) \\ \textsc{MaxSubarray}(a[\frac{n}{2}, \ldots, n]) \\ \textsc{MaxSuffix}(a[1, \ldots, \frac{n}{2}]) + \textsc{MaxPrefix}(a[\frac{n}{2}, \ldots, n]) \end{cases}$$

**analysis** $(O(n)$ time for non-recursive work$) \times (O(\log n)$ depth$) = O(n \log n)$

## recursion inversion for max subarray

the max subarray either uses the last element or doesn't:

$$\text{MaxSubarray}(a[1,\ldots,n]) = \max\begin{cases}\text{MaxSubarray}(a[1,\ldots,n-1])\\ \text{MaxSuffix}(a[1,\ldots,n]\end{cases},$$

$$\text{MaxSuffix}(a[1,\ldots,n]) = \max\{0, \text{MaxSuffix}(a[1,\ldots,n-1]) + a[n]\}$$

**dynamic programming** evaluate this non-recursively by computing

- first MaxSubarray($a[1]$) and MaxSuffix($a[1]$)
- then MaxSubarray($a[1,2]$) and MaxSuffix($a[1,2]$) from above
- then MaxSubarray($a[1,2,3]$) and MaxSuffix($a[1,2,3]$) from above
- and so on

**analysis** computing MaxSubarray($a[1,\ldots,n]$) and MaxSuffix($a[1,\ldots,n]$
from MaxSubarray($a[1,\ldots,n-1]$) and MaxSuffix($a[1,\ldots,n-1]$)
takes $O(1)$ time

$O(n)$ things to compute $= O(n)$ time

# does algorithm design matter?

**TABLE I. Summary of the Algorithms**

| Algorithm | | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Lines of C Code | | 8 | 7 | 14 | 7 |
| Run time in microseconds | | $3.4N^3$ | $13N^2$ | $46N \log N$ | $33N$ |
| Time to solve problem of size | $10^2$ | 3.4 secs | 130 msecs | 30 msecs | 3.3 msecs |
| | $10^3$ | .94 hrs | 13 secs | .45 secs | 33 msecs |
| | $10^4$ | 39 days | 22 mins | 6.1 secs | .33 secs |
| | $10^5$ | 108 yrs | 1.5 days | 1.3 min | 3.3 secs |
| | $10^6$ | 108 mill | 5 mos | 15 min | 33 secs |
| Max problem solved in one | sec | 67 | 280 | 2000 | 30,000 |
| | min | 260 | 2200 | 82,000 | 2,000,000 |
| | hr | 1000 | 17,000 | 3,500,000 | 120,000,000 |
| | day | 3000 | 81,000 | 73,000,000 | 2,800,000,000 |

Digital Equipment Corporation VAX-11/750 in 1984

## what if my problem is not in P?

find something else in polynomial time:

- a solution close to optimal *(approximate)*
- an optimal solution in expectation *(average-case analysis)*
- solutions to problems with particularly good solutions *(planted analyses)*
- solutions that are small *(parameterized analysis)*
- solutions to *nice* instances *(smoothed analysis)*
- a locally optimal solutions *(local search)*

or you could use a *heuristic* and not guarantee anything
or you could spend exponential time and have patience

## what if I don't know if my problem is in P or is NP-hard?

your problem could be NP-intermediate
such as:

- comparing sums of square roots
- integer factorization
- computing the discrete logarithm