Computational Complexity II: Asymptotic Notation and Classification Algorithms

Maria-Eirini Pegia

Seminar on Theoretical Computer Science and Discrete Mathematics
Aristotle University of Thessaloniki

Context

Computational Complexity

- Computational Complexity of an algorithm A:

Computational Complexity

- Computational Complexity of an algorithm A:
  - ♢ time, space (memory)

Computational Complexity

- Computational Complexity of an algorithm A:
  - ♢ time, space (memory)
  - ♢ worst, average, best case

Computational Complexity

- Input snapshot size n:

Computational Complexity

- Input snapshot size n:
  - ♢ #bits for the representation input data to the memory

Computational Complexity

- Input snapshot size n:
  - ♢ #bits for the representation input data to the memory
  - ♢ number of basic components which constitute the size and difficulty measure of snapshot
    (i.e., vetrices and edges of a graph)

Computational Complexity

- Computational Complexity of the problem P:

Computational Complexity

- Computational Complexity of the problem P:

  Complexity (worst case) best algorithm which solves the problem P.

Algorithm Design

- Valuation of computational complexity

Algorithm Design

- Valuation of computational complexity
  - ♢ worst case

Algorithm Design

- Valuation of computational complexity
  - ♢ worst case

  Running time guarantees for any input of size n.

Algorithm Design

- Valuation of computational complexity

  ◊ **worst** case

  Running time guarantees for any input of size n.
  - Generally captures efficiency in practice .
  - ◊ Draconian view, but hard to find effective alternative

Algorithm Design

- ♢ **average** case

Algorithm Design

- ♢ **average** case

the performance of an algorithm averaged over **"random"** instances can sometimes provide considerable insight.

Algorithm Design

- ♢ best case

Algorithm Design

- **best** case

The term best case performance is used to describe an algorithm's behavior under optimal conditions.

Algorithm Design

⋄ best case

The term best case performance is used to describe an algorithm's behavior under optimal conditions.

- Best solution depending on application requirements.

Algorithm Design

⋄ best case

The term best case performance is used to describe an algorithm's behavior under optimal conditions.

- Best solution depending on application requirements.
- Average performance and worst-case performance are the most used in algorithm analysis.

Asymptotic Valuation

- Running time of the algorithm A:
  ♢ Increasing function of T(n) that expresses in how much time is completed A when is applied in snapshot of size n.

Asymptotic Valuation

- Running time of the algorithm A:
  ♢ Increasing function of T(n) that expresses in how much time is completed A when is applied in snapshot of size n.

- We are interested in the size class T(n)
  ♢ Size class is intrinsic property of algorithm.

Asymptotic Valuation

- Running time of the algorithm A:
  ♢ Increasing function of T(n) that expresses in how much time is completed A when is applied in snapshot of size n.

- We are interested in the size class T(n)
  ♢ Size class is intrinsic property of algorithm.
  - binary search → logarithmic time
  - dynamic programming → linear time

Asymptotic Valuation

- Running time of the algorithm A:
  ♢ Increasing function of T(n) that expresses in how much time is completed A when is applied in snapshot of size n.

- We are interested in the size class T(n)
  ♢ Size class is intrinsic property of algorithm.
  - binary search → logarithmic time
  - dynamic programming → linear time

- ignores stables & focuses on runtime size class

Asymptotic Upper Bounds

Big-Oh notation

T(n) is $O(|f(n)|)$ if there exist constants c > 0 and $n_0 \geq 0$ such that $T(n) \leq c \cdot f(n)$ $\forall$ $n \geq n_0$.

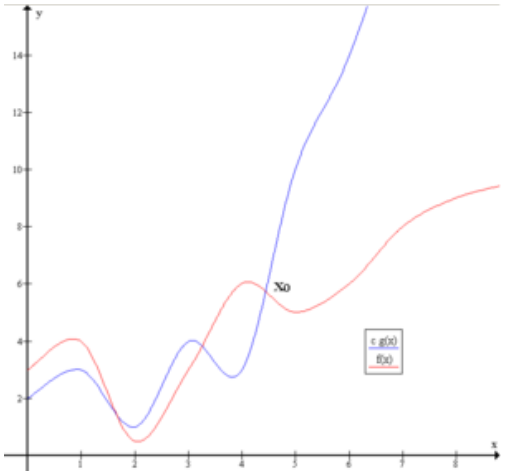


Figure: f(x) $\in$ O(g(x))

Big-Oh notation

Example

$T(n) = pn^2 + qn + r, \quad p,q,r > 0$

Big-Oh notation

Example

$T(n) = pn^2 + qn + r, \quad p,q,r > 0$

We claim that any such function is $O(n^2)$.

Big-Oh notation

Example

$T(n) = pn^2 + qn + r, \quad p,q,r > 0$

We claim that any such function is $O(n^2)$.

$\forall\ n \geq 1,\ qn \leq qn^2$ and $r \leq rn^2$

$T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 = (p+q+r)\ n^2 = c\ n^2$

Big-Oh notation

♢ O(f(n)) is a set of functions

- $T(n) \in O(f(n))$ (✓)
- $T(n) = O(f(n))$ (x, ✓)

Big-Oh notation

♢ O(f(n)) is a set of functions

- $T(n) \in O(f(n))$ (✓)
- $T(n) = O(f(n))$ (x, ✓)

♢ Nonnegative functions: When using Big-Oh notation, we assume that the functions involved are (asymptotically) nonnegative.

Asymptotic Lower Bounds

Big-Omega notation

T(n) is $\Omega(\ f(n))$ if there exist constants c > 0 and $n_0 \geq 0$ such that $T(n) \geq c \cdot f(n)\ \forall\ n \geq n_0$.

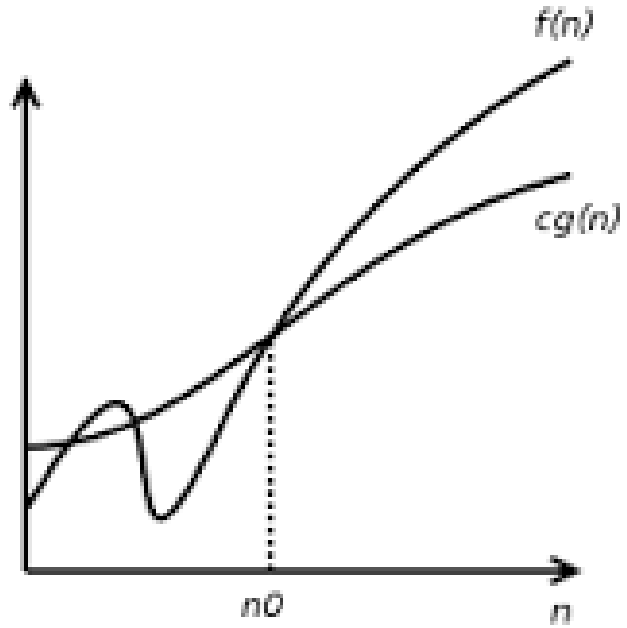


Figure: Big-Omega notation

Asymptotic Lower Bounds

Example

$T(n) = pn^2 + qn + r, \quad p,q,r > 0$

Asymptotic Lower Bounds

Example

$T(n) = pn^2 + qn + r, \quad p,q,r > 0$

Let's claim that $T(n) = \Omega(n^2)$.

Asymptotic Lower Bounds

Example

$T(n) = pn^2 + qn + r, \quad p,q,r > 0$

Let's claim that $T(n) = \Omega(n^2)$.

$\forall\ n \geq 0,\ T(n) = pn^2 + qn + r \geq pn^2 = c\ n^2$

Asymptotically Tight Bounds

Big-Theta notation

T(n) is $\Theta(\ f(n)\ )$ if there exist constants $c_1 > 0$, $c_2 > 0$ and $n_0 \geq 0$ such that

$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$

$\forall\ n \geq n_0$.

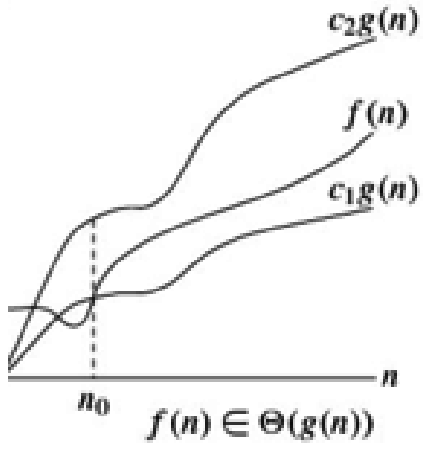


Figure: Big-Theta notation

Big-Theta notation

Example

$T(n) = pn^2 + qn + r, \quad p,q,r > 0$

Big-Theta notation

Example

$T(n) = pn^2 + qn + r, \quad p,q,r > 0$

We saw that $T(n)$ is both $O(n^2)$ and $\Omega(n^2)$.

$T(n) = \Theta(n^2)$

Relational Properties

- Reflexivity: O, Ω, Θ

Relational Properties

- Reflexivity: O, Ω, Θ
- Transitivity: O, Ω, Θ

Relational Properties

- Reflexivity: O, Ω, Θ
- Transitivity: O, Ω, Θ
- Symmetry: $f(n) = \Theta(\ g(n)\ ) \iff g(n) = \Theta(\ f(n)\ )$

Relational Properties

- Reflexivity: O, Ω, Θ
- Transitivity: O, Ω, Θ
- Symmetry: $f(n) = \Theta(\ g(n)\ ) \iff g(n) = \Theta(\ f(n)\ )$
- Transpose Symmetry (Duality):

$$f(n) = O(\ g(n)\ ) \iff g(n) = \Omega(\ f(n)\ )$$

Master Theorem

Master Theorem

If T(n) = a $T(\frac{n}{b})$ + O(n^d) for constants a > 0, b > 1, d ≥ 0, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > log_b a \\ O(n^d logn) & \text{if } d = log_b a \\ O(n^{log_b a}) & \text{if } d < log_b a \end{cases}$$

Nice Trick for computing quickly the computational complexity.

Notation	Name
O(1)	constant
O(loglogn)	double logarithmic
O(logn)	logarithmic
O(n)	linear
O(nlogn)	loglinear
$O(n^2)$	quadratic
$O(n^c)$, c>1	polynomial
$O(e^n)$	exponential
O(n!)	factorial

Efficient algorithm

Definition 1

An algorithm is efficient if, when implemented, it runs quickly on real input instances.

Efficient algorithm

Definition 1

An algorithm is efficient if, when implemented, it runs quickly on real input instances.

♢ the omission of where, and how well

♢ real input instances

♢ scale

Efficient algorithm

Definition 2

An algorithm is efficient if it achieves qualitatively better worst case performance, at an analytical level, than brute-force search.

Efficient algorithm

Definition 2

An algorithm is efficient if it achieves qualitatively better worst case performance, at an analytical level, than brute-force search.

♢ What do we mean by "qualitatively better performance"?

Efficient algorithm

Definition 2

An algorithm is efficient if it achieves qualitatively better worst case performance, at an analytical level, than brute-force search.

♢ What do we mean by "qualitatively better performance"?

Definition 3

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

Poly-time algorithm

♢ Desirable scaling property: When the input size doubles, the algorithm should only slow down by some constant factor C.

Poly-time algorithm

♢ Desirable scaling property: When the input size doubles, the algorithm should only slow down by some constant factor C.

Definition

An algorithm is poly-time if the above scaling property holds.

Poly-time algorithm

♢ Desirable scaling property: When the input size doubles, the algorithm should only slow down by some constant factor C.

Definition

An algorithm is poly-time if the above scaling property holds.

♢ There exists constants c > 0 and d > 0 such that on every input of size n, its running time is bounded by cn^d primitive computational steps.

Why we care for the asymptotic bound of an algorithm?

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Classification algorithms by their design methodology

- Brute-force (or exhaustive search)

Classification algorithms by their design methodology

- Brute-force (or exhaustive search)
- Divide and conquer

Classification algorithms by their design methodology

- Brute-force (or exhaustive search)
- Divide and conquer
- Dynamic programming

Classification algorithms by their design methodology

- Brute-force (or exhaustive search)
- Divide and conquer
- Dynamic programming
- Randomized algorithms

Brute-force search

Definition

Brute-force search is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

Brute-force search

Definition

Brute-force search is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

♢ (-) can end up doing far more work to solve a given problem than might do a more clever or sophisticated algorithm

Brute-force search

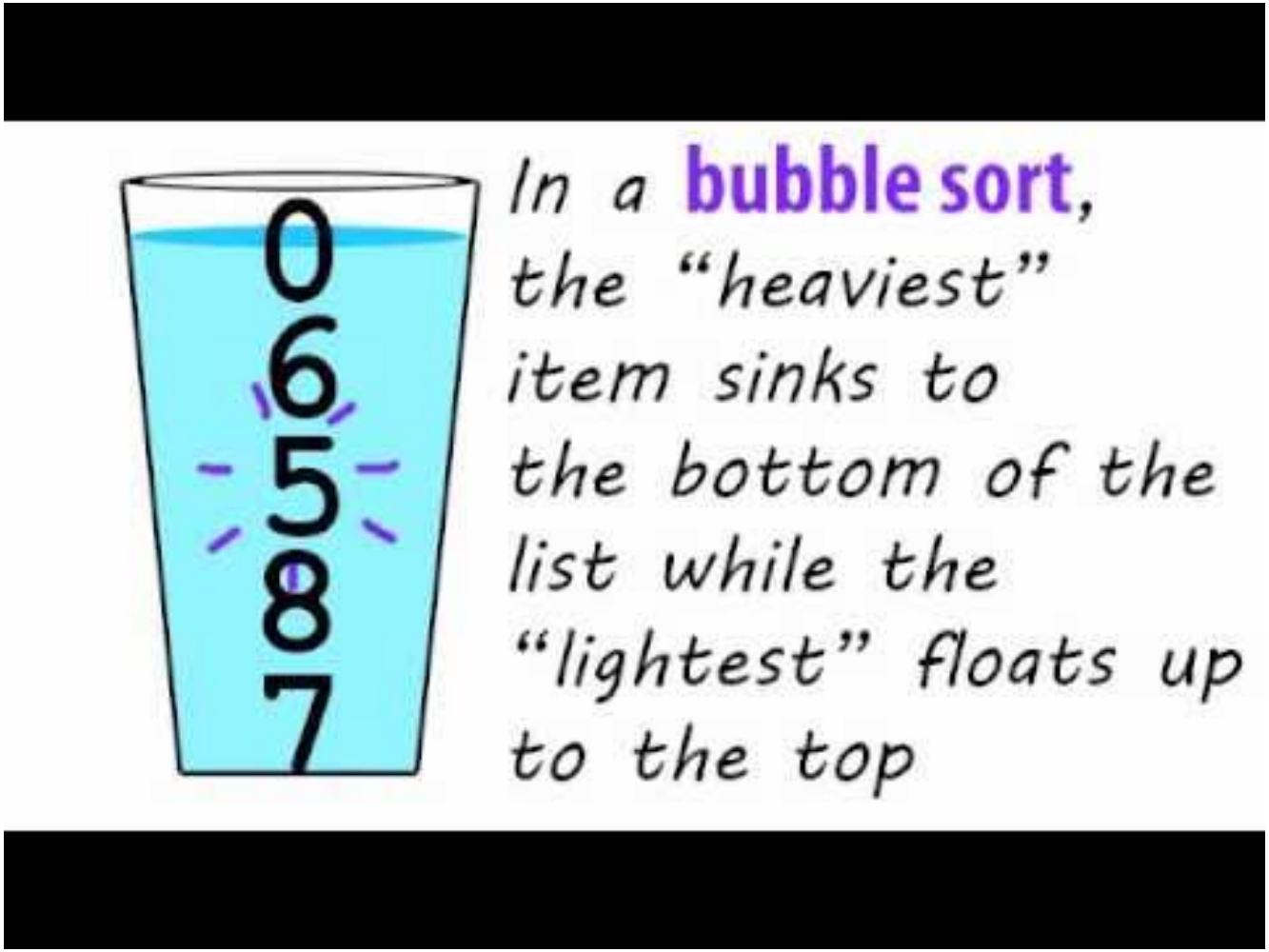
Definition

Brute-force search is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

♢ (-) can end up doing far more work to solve a given problem than might do a more clever or sophisticated algorithm

♢ (+) is often easier to implement than a more sophisticated one, because of this simplicity, sometimes it can be more efficient

Bubble Sort


0
6
5
8
7
In a bubble sort,
the "heaviest"
item sinks to
the bottom of the
list while the
"lightest" floats up
to the top

Bubble Sort

Input: array a with n elements

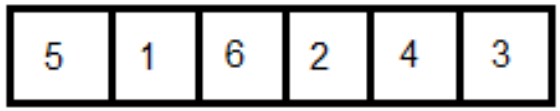
```
For i = 1    to    n

    For j = i + 1    to    n

        If ( a[i] > a[j] )
            then swap their values
        End - If

    End - For

End - For
```



Bubble Sort

- Computational Complexity: $O(n^2)$

Bubble Sort

- Computational Complexity: $O(n^2)$

WHY???

Bubble Sort

♢ Computational Complexity: $O(n^2)$

WHY???

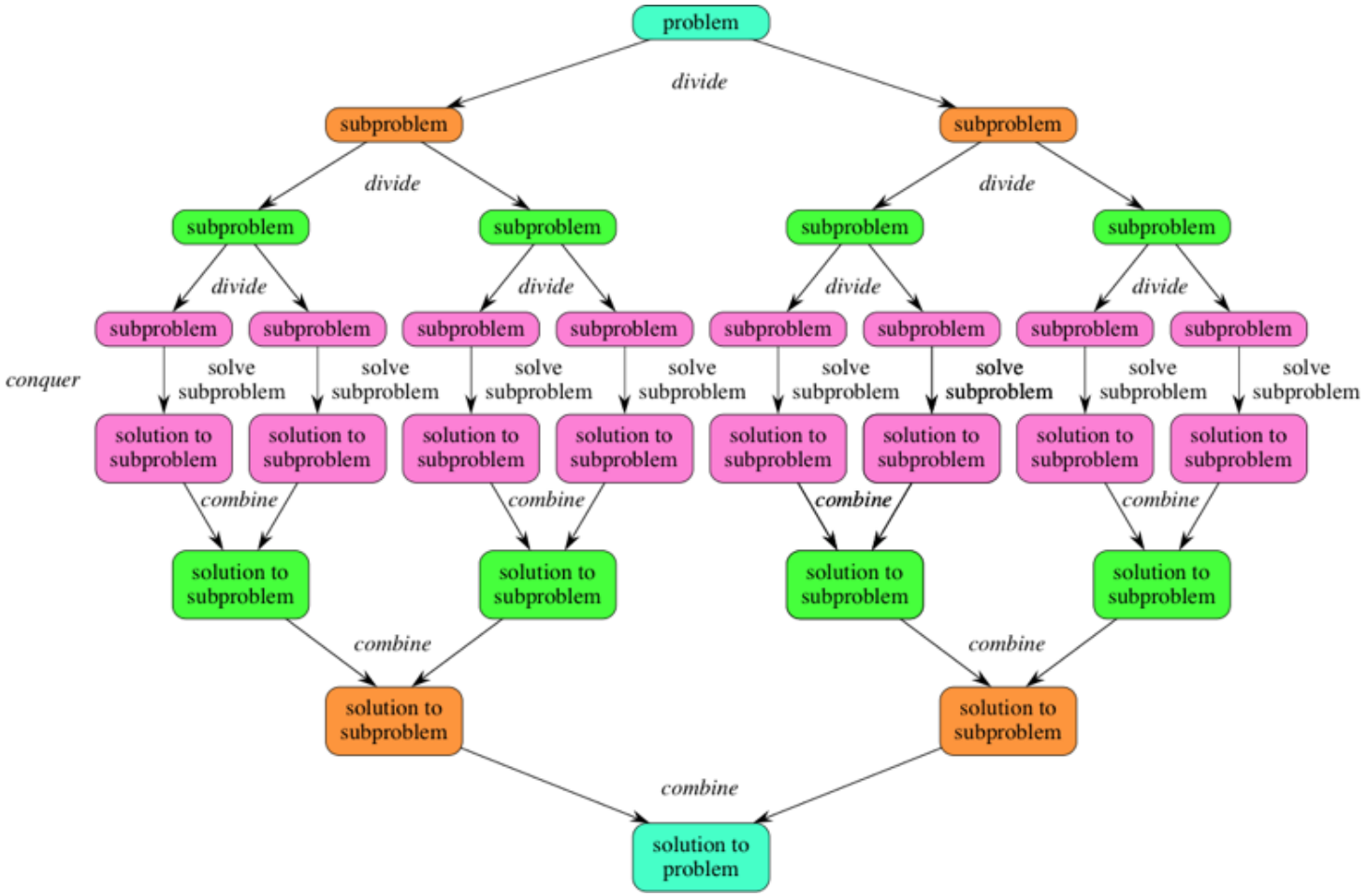
♢ not a practical sorting algorithm when n is large

Divide and Conquer

Definition

A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

3 steps of divide and conquer



Mergesort

Example

$T(n)$ = number of comparisons to mergesort an input of size n.

Mergesort

Example

$T(n)$ = number of comparisons to mergesort an input of size n.

- Divide array into two halves (divide $O(1)$).

Mergesort

Example

$T(n)$ = number of comparisons to mergesort an input of size n.

- Divide array into two halves (divide $O(1)$).
- Recursively sort each half (sort $2T(\frac{n}{2})$).

Mergesort

Example

$T(n)$ = number of comparisons to mergesort an input of size n.

- Divide array into two halves (divide $O(1)$).
- Recursively sort each half (sort $2T(\frac{n}{2})$).
- Merge two halves to make sorted whole (merge $O(n)$).

Mergesort

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(\frac{n}{2}) + O(n) & \text{otherwise} \end{cases}$$

Mergesort

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(\frac{n}{2}) + O(n) & \text{otherwise} \end{cases}$$

- ♢ Master Theorem ☺

Mergesort

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(\frac{n}{2}) + O(n) & \text{otherwise} \end{cases}$$

- ♢ Master Theorem ☺
- ♢ Solution: $T(n) = O(n log(n))$

Dynamic programming

Divide and Conquer

Break up a problem into independent sub-problems, solve each sub-problem, and combine solution to sub-problems to form solution to original problem.

Dynamic programming

Divide and Conquer

Break up a problem into independent sub-problems, solve each sub-problem, and combine solution to sub-problems to form solution to original problem.

Dynamic programming

Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Applications

- Dynamic programming applications:

Applications

- Dynamic programming applications:

  Bioinformatics

Applications

- Dynamic programming applications:

  Bioinformatics

  Control Theory

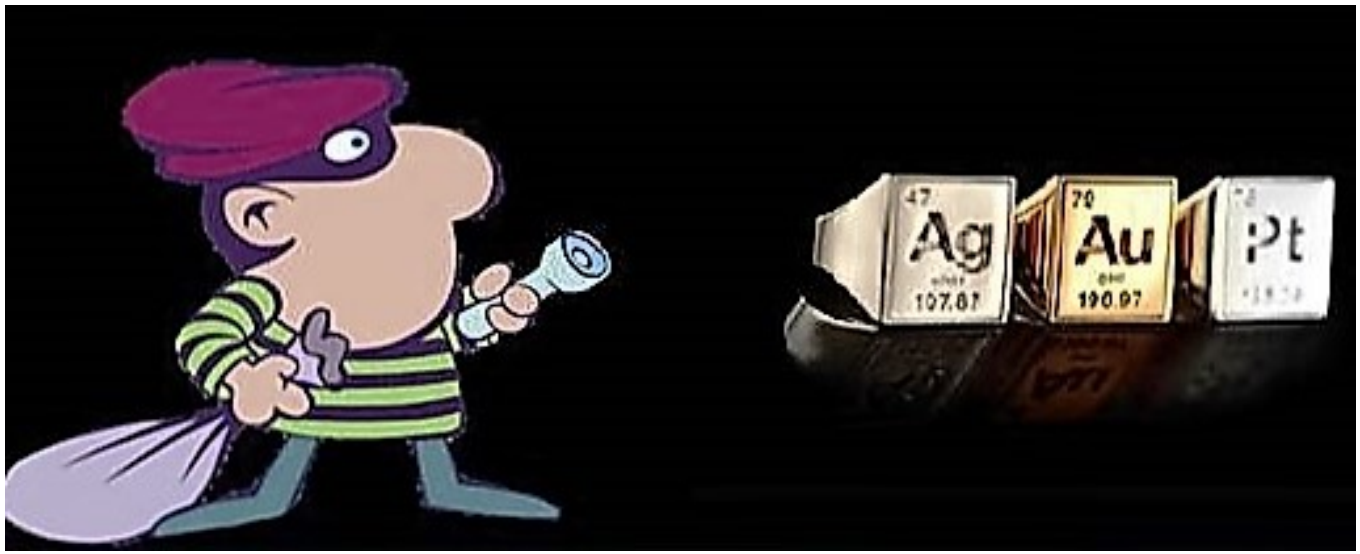
Applications

- Dynamic programming applications:
  - Bioinformatics
  - Control Theory
  - Information Theory

Applications

- Dynamic programming applications:
  - Bioinformatics
  - Control Theory
  - Information Theory
- Cocke-Kasami-Younger for parsing context-free grammars.

Knapsack Problem



Knapsack

- Given n objects and a "knapsack".

Knapsack

- Given n objects and a "knapsack".
- Item i weights $w_i > 0$ and has value $v_i > 0$.

Knapsack

- Given n objects and a "knapsack".
- Item i weights $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of W.

Knapsack

- Given n objects and a "knapsack".
- Item i weights $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of W.
- **Goal**: fill knapsack so as to maximize total value.

Knapsack problem

OPT(i, w) = max profit subset of items 1, . . . , i with weight limit w

Knapsack problem

OPT(i, w) = max profit subset of items 1, . . . , i with weight limit w

♢ Case 1: OPT does not select item i.

Knapsack problem

OPT(i, w) = max profit subset of items 1, . . . , i with weight limit w

♢ Case 1: OPT does not select item i.

OPT selects best of { 1, 2, . . . , i – 1 } using weight limit w.

Knapsack problem

OPT(i, w) = max profit subset of items 1, . . . , i with weight limit w

◊ Case 1: OPT does not select item i.

OPT selects best of { 1, 2, . . . , i – 1 } using weight limit w.

◊ Case 2: OPT selects item i.

Knapsack problem

OPT(i, w) = max profit subset of items 1, . . . , i with weight limit w

♢ Case 1: OPT does not select item i.

OPT selects best of { 1, 2, . . . , i – 1 } using weight limit w.

♢ Case 2: OPT selects item i.

New weight limit = w – w_i.

Knapsack problem

OPT(i, w) = max profit subset of items 1, . . . , i with weight limit w

◊ Case 1: OPT does not select item i.

OPT selects best of { 1, 2, . . . , i – 1 } using weight limit w.

◊ Case 2: OPT selects item i.

New weight limit = w – w_i.

OPT selects best of { 1, 2, . . . , i – 1 } using this new weight limit.

Knapsack problem

Objective function:

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w), v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Knapsack Problem

♢ Knapsack problem: bottom-up

KNAPSACK (n, W, $w_1, \ldots, w_n, v_1, \ldots, v_n$)
for w = 0 to W
    M [0, w] ⟵ 0.
for i = 1 to n
    for w = 0 to W
      if (w_i > w)
          M [i, w] ⟵ M [i – 1, w].
      else
          M [i, w] ⟵ max{ M [i – 1, w], v_i + M [i – 1, w – w_i] }
return M [n, W]

Knapsack problem: running time

Theorem

There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(nW)$ time and $\Theta(nW)$ space.

Knapsack problem: running time

Theorem

There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(nW)$ time and $\Theta(nW)$ space.

♢ not polynomial in input size! (pseudo-polynomial)

Knapsack problem: running time

Theorem

There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(nW)$ time and $\Theta(nW)$ space.

♢ not polynomial in input size! (pseudo-polynomial)

♢ NP - complete problem ☺

Randomized Algorithms

Definition

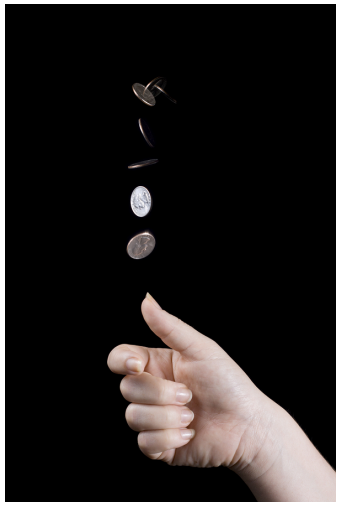
A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic.

Randomized Algorithms

Definition

A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic.

♢ Randomization: Allow fair coin flip in unit time.



Randomized Algorithms

Why randomize?

Randomized Algorithms

Why randomize?

Can lead to simplest, fastest, or only known algorithm for a particular problem! ☺

Randomized Algorithms

There are two large classes of such algorithms:

Randomized Algorithms

There are two large classes of such algorithms:

- **Las Vegas**: A randomized algorithm that always outputs the correct answer, it is just that there is a small probability of taking long to execute.

Randomized Algorithms

There are two large classes of such algorithms:

⋄ **Las Vegas**: A randomized algorithm that always outputs the correct answer, it is just that there is a small probability of taking long to execute.

⋄ **Monte Carlo**: Sometimes we want the algorithm to always complete quickly, but allow a small probability error.

Randomized Algorithms

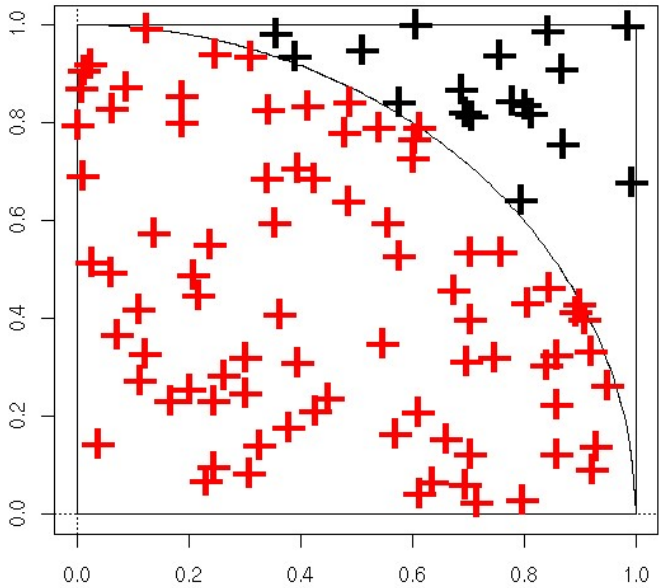
Any Las Vegas algorithm can be converted into a Monte Carlo algorithm by outputting an arbitrary, possibly incorrect answer if it fails to complete within a specified time.

Randomized Algorithms

Any Las Vegas algorithm can be converted into a Monte Carlo algorithm by outputting an arbitrary, possibly incorrect answer if it fails to complete within a specified time.

Monte Carlo algorithm cannot be converted into a Las Vegas (i.e., approximation of π)

Monte Carlo vs Las Vegas



$$\pi \approx 4 \frac{n_{(\frac{cycle}{4})}}{n_{(square)}}$$

Next

- ♢ Computational Complexity
- ♢ Complexity Classes (i.e., $\mathcal{P}$, $\mathcal{NP}$)
- ♢ Some nice computational problems ☺
- ♢ Some reductions

\(•‿•)/
\(•‿•)/ \(•‿•)/
\(•‿•)/ \(•‿•)/ \(•‿•)/
\(•‿•)/ \(•‿•)/ \(•‿•)/ \(•‿•)/
\(•‿•)/ \(•‿•)/ \(•‿•)/ \(•‿•)/ \(•‿•)/

References

- J.Kleinberg, E.Tardos. Algorithm Design. Boston, Mass.: Pearson/Addison-Wesley, cop. 2006
- Ι.Μανωλόπουλος, Α.Παπαδόπουλος, Κ.Τσίχλας. Θεωρία και Αλγόριθμοι Γράφων, Αθήνα: Εκδ. Νέων Τεχνολογιών, 2014.
- Τσίχλας, Κ., Γούναρης, Α., Μανωλόπουλος, Ι., 2015. Σχεδίαση και ανάλυση αλγορίθμων. [ηλεκτρ. βιβλ.] Αθήνα:Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών. Διαθέσιμο στο: http:\\hdl.handle.net\11419\4005
- Δομές δεδομένων, Μποζάνης Παναγιώτης Δ, ΕΚΔΟΣΕΙΣ Α. ΤΖΙΟΛΑ & ΥΙΟΙ Α.Ε., 2006

Thank you!!!