2 - Algorithm Complexity

Charles A. Wuethrich

Bauhaus-University Weimar
CoGVis/MMC

April 24, 2019

Algorithm Complexity

- Maybe we have more than one algorithm solving one problem.
- We need a way to compare speed among them.
  - Can use execution timing?
  - No, because it depends on machine

  - Maybe then counting operations?
  - Their number depends on data
  - Besides, which operations should one count?
    Small remark on mathematicians...

  - Speed depends on data size
  - Thus its speed needs to filter it out

Algorithm Complexity

- Maybe we have more than one algorithm solving one problem.
- We need a way to compare speed among them.
  - Can use execution timing?
  - No, because it depends on machine

  - Maybe then counting operations?
  - Their number depends on data
  - Besides, which operations should one count?
    Small remark on mathematicians...

  - Speed depends on data size
  - Thus its speed needs to filter it out

Algorithm Complexity

- Maybe we have more than one algorithm solving one problem.
- We need a way to compare speed among them.
  - Can use execution timing?
  - No, because it depends on machine

  - Maybe then counting operations?
  - Their number depends on data
  - Besides, which operations should one count?
    Small remark on mathematicians...

  - Speed depends on data size
  - Thus its speed needs to filter it out

Algorithm Complexity

- Maybe we have more than one algorithm solving one problem.
- We need a way to compare speed among them.
  - Can use execution timing?
  - No, because it depends on machine

  - Maybe then counting operations?
  - Their number depends on data
  - Besides, which operations should one count?
    Small remark on mathematicians...

  - Speed depends on data size
  - Thus its speed needs to filter it out

Algorithm Complexity

- Maybe we have more than one algorithm solving one problem.
- We need a way to compare speed among them.
    - Can use execution timing?
    - No, because it depends on machine

    - Maybe then counting operations?
    - Their number depends on data
    - Besides, which operations should one count?
      Small remark on mathematicians...

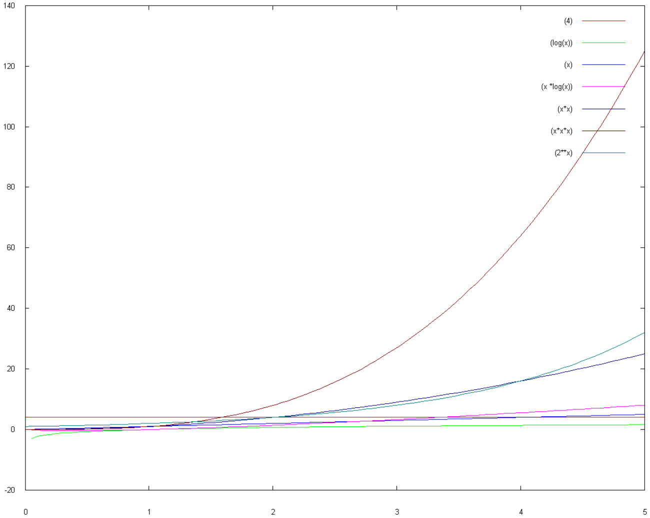
    - Speed depends on data size
    - Thus its speed needs to filter it out

Algorithm Complexity

- Suppose N is the size of the data to be processed.
  - Constant time: The instructions of the algorithm are a finite number and independent from amount of data.
  - Logarithmic time: The instructions of the algorithm are repeated an $\mathcal{O}(\log N)$ number of times or a finite number of times.
  - Linear time: The instructions of the algorithm are repeated an $\mathcal{O}(N)$ number of times or less.
  - $N \log N$ time: The instructions of the algorithm are repeated an $\mathcal{O}(N \log N)$ number of times or less.
  - Quadratic time: The instructions of the algorithm are repeated an $\mathcal{O}(N^2)$ number of times or less.
  - Cubic time: The instructions of the algorithm are repeated an $\mathcal{O}(N^3)$ number of times or less.
  - Exponential time: The instructions of the algorithm are repeated an $\mathcal{O}(a^N)$ number of times or less.

Complexity

- The resulting functions look as follows:



Complexity: definition and computations

- If the program execution time $X \geq 0$ is such that for a given positive function $f(N)$, $\lim_{N \to \infty} \frac{X}{f(N)} \leq K$ for some $K > 0$, then execution time is $\mathcal{O}(f(N))$.

- When an algorithm can be decomposed in more parts, the arithmetic of complexity is like computations at infinity.
  - If the algorithm has 3 consecutive steps, running in $\mathcal{O}(f_1(N))$, $\mathcal{O}(f_2(N))$, and $\mathcal{O}(f_3(N))$, $\Longrightarrow$ its complexity is $\mathcal{O}(|f_1(N)| + |f_2(N)| + |f_3(N)|) = Max_i |f_i(N)|$, for $N > N_0$.
  - If the algorithm has 2 nested steps, running in $\mathcal{O}(f_1(N))$ and $\mathcal{O}(f_2(N))$, $\Longrightarrow$ its complexity is $\mathcal{O}(|f_1(N)| \cdot |f_2(N)|)$.
  - For a proof, just unroll the limit definition.

Complexity: hands on with sorting

- This sounds very abstract: let us make a good example: sorting

- What is sorting? order a given set according to an order relation.
- What is an order relation?
- It is a relation in a set S among pairs of elements such that it satisfies the following properties:
    1. reflexive: $\forall a \in S, a \leq a$,
    2. antisymmetric: $a \leq b$ and $b \leq a \iff a \equiv b$,
    3. transitive: $a \leq b$ and $b \leq c \implies a \leq c$.
- Example: telephone directory uses lexicographical order

Sorting

- The sorting problem:

  Given an array of N data
  $\Longrightarrow$ Order the array elements according to given order relation

- Basic method used: compare and swap two elements
- Efficiency: depends on speed and memory usage
  - Speed: can be estimated through the number of compares and swaps
  - Memory usage: Here not relevant due to work "on the array".
- A method is said to be stable if the relative position of elements with he same key is not changed.
  $1_a, 2_a, 1_b, 2_b, 2_c, 2_d, 3 \Longrightarrow 1_a, 1_b, 2_a, 2_b, 2_c, 2_d, 3$

Selection sort

- Sort through direct selection
- Idea:
    - Look for smallest element
    - Swap with first element
    - Move to second element
    - Repeat with rest of array
- Speed: How many swaps and compares do I do?
- 1st run: Max $(N-1)$ compares and 1 swap.
- 2nd run: Max $(N-2)$ compares and 1 swap . . .
- . . . till the vector is finished.
- Now, the sum of the first $(N-1)$ integers equals $n*(n-1)/2$, thus the whole becomes
- Total: $\frac{(N)(N-1)}{2} \propto N^2 \Longrightarrow \mathcal{O}(N^2)$.

Selection sort

- Sort through direct selection
- Idea:
  - Look for smallest element
  - Swap with first element
  - Move to second element
  - Repeat with rest of array
- Speed: How many swaps and compares do I do?
- 1st run: Max $(N-1)$ compares and 1 swap.
- 2nd run: Max $(N-2)$ compares and 1 swap ...
- ... till the vector is finished.
- Now, the sum of the first $(N-1)$ integers equals $n*(n-1)/2$, thus the whole becomes
- Total: $\frac{(N)(N-1)}{2} \propto N^2 \implies \mathcal{O}(N^2)$.

Selection sort

- Sort through direct selection
- Idea:
  - Look for smallest element
  - Swap with first element
  - Move to second element
  - Repeat with rest of array
- Speed: How many swaps and compares do I do?
- 1st run: Max $(N-1)$ compares and 1 swap.
- 2nd run: Max $(N-2)$ compares and 1 swap …
- …till the vector is finished.
- Now, the sum of the first $(N-1)$ integers equals $n*(n-1)/2$, thus the whole becomes
- Total: $\frac{(N)(N-1)}{2} \propto N^2 \Longrightarrow \mathcal{O}(N^2)$.

Insertion sort

- Sort through insertion
- Idea:
  - Choose a still unsorted element (first time: first element)
  - Put it in the right place among already sorted elements
  - Take next unsorted element until whole array is sorted
- Speed:
  - Average case: Depends on order of elements.
  - Worst case: Take all N elements, and insert them as last (i.e. compare with $0, 1, 2, \ldots, N-1$
    $\frac{N(N-1)}{2}$ compares.
- Total: $\mathcal{O}(N^2)$.
- BUT: When data almost sorted: $\mathcal{O}(N)$.

Insertion sort

- Sort through insertion
- Idea:
  - Choose a still unsorted element (first time: first element)
  - Put it in the right place among already sorted elements
  - Take next unsorted element until whole array is sorted
- Speed:
  - Average case: Depends on order of elements.
  - Worst case: Take all N elements, and insert them as last (i.e. compare with $0, 1, 2, \ldots, N-1$ $\frac{N(N-1)}{2}$ compares.
- Total: $\mathcal{O}(N^2)$.
- BUT: When data almost sorted: $\mathcal{O}(N)$.

Insertion sort

- Sort through insertion
- Idea:
  - Choose a still unsorted element (first time: first element)
  - Put it in the right place among already sorted elements
  - Take next unsorted element until whole array is sorted
- Speed:
  - Average case: Depends on order of elements.
  - Worst case: Take all N elements, and insert them as last (i.e. compare with $0, 1, 2, \ldots, N-1$ $\frac{N(N-1)}{2}$ compares.
- Total: $\mathcal{O}(N^2)$.
- BUT: When data almost sorted: $\mathcal{O}(N)$.

Bubble sort

- Sort through swapping
- Idea:
  - Move through array from left to right
  - Swap elements if they are in wrong order
  - Repeat until there is nothing left to swap.
- Speed:
  - At each step i: $N - i$ compares, and $\leq N - i$ swaps.
  - At most N steps (last moved to first).
- Total: $\mathcal{O}(N^2)$.
- BUT: When data almost sorted: $\mathcal{O}(N)$.

Bubble sort

- Sort through swapping
- Idea:
  - Move through array from left to right
  - Swap elements if they are in wrong order
  - Repeat until there is nothing left to swap.
- Speed:
  - At each step i: $N - i$ compares, and $\leq N - i$ swaps.
  - At most N steps (last moved to first).
- Total: $\mathcal{O}(N^2)$.
- BUT: When data almost sorted: $\mathcal{O}(N)$.

Bubble sort

- Sort through swapping
- Idea:
  - Move through array from left to right
  - Swap elements if they are in wrong order
  - Repeat until there is nothing left to swap.
- Speed:
  - At each step i: $N - i$ compares, and $\leq N - i$ swaps.
  - At most N steps (last moved to first).
- Total: $\mathcal{O}(N^2)$.
- BUT: When data almost sorted: $\mathcal{O}(N)$.

Shell sort

- Improved insertion sort (Donald Shell)
- Idea:
  - We subdivide into subsets (like a matrix) by picking the s_n-th element $s_0 = 1,\ s_{i+1} = 3s_i + 1$
    $\Longrightarrow \{1, 4, 13, 40, 121, 364, \ldots\} = S$
  - Order elements in each subset with insertion sort
  - Resubdivide into new subsets by picking s_{n-1}-th element.
  - Order elements in each new subset with insertion sort
  - The S_n sequence works always as long as last sort step is one
- Speed:
  - The speed is dependent on the sequence s_i used[1]..
- Worst case: $\mathcal{O}(N^{3/2})$ for sequence $\{1, 4, 13, 40, 121, 364, \ldots\}$.
- Best case: $\geq N^{1+c/\sqrt{M}}$ compares for N elements in M passes.
- Average case for S: possibly $\mathcal{O}(N^{5/4})$ or $\mathcal{O}(N \log N^2)$

---

[1] For details: http://www.cs.princeton.edu/ rs/shell/paperF.pdf

Shell sort

- Improved insertion sort (Donald Shell)
- Idea:
  - We subdivide into subsets (like a matrix) by picking the s_n-th element $s_0 = 1,\ s_{i+1} = 3s_i + 1$ $\Longrightarrow \{1, 4, 13, 40, 121, 364, \ldots\} = S$
  - Order elements in each subset with insertion sort
  - Resubdivide into new subsets by picking s_{n-1}-th element.
  - Order elements in each new subset with insertion sort
  - The S_n sequence works always as long as last sort step is one
- Speed:
  - The speed is dependent on the sequence s_i used[1]..
- Worst case: $\mathcal{O}(N^{3/2})$ for sequence $\{1, 4, 13, 40, 121, 364, \ldots\}$.
- Best case: $\geq N^{1+c/\sqrt{M}}$ compares for N elements in M passes.
- Average case for S: possibly $\mathcal{O}(N^{5/4})$ or $\mathcal{O}(N \log N^2)$

---

[1] For details: http://www.cs.princeton.edu/ rs/shell/paperF.pdf

Shell sort

14	34	8	72	1	45	29	19	75	Das unsortierte Feld
14	34	8	72	1	45	29	19	75	Wird in Teilfelder zerlegt (Abstand=3)
14	1	8	29	19	45	72	34	75	die Teilfelder sortieren
14	1	8	29	19	45	72	34	75	Weiter zerteilen (Abstand =1) und sortieren
1	8	14	19	29	34	45	72	75	und fertig!

- This also shows how speedy the algorithm is.

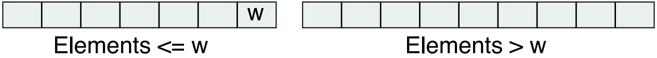
Comparison

	SelectionSort	InsertionSort	ShellSort
Comparisons Ave.; Worst	$N^2/2$; $N^2/2$	$N^2/2$; $N^2/2$	$N^{5/4}$; $N^{3/2}$
Comparisons Ave.; Worst	N; N	$N^2/8$; $N^2/4$	$N^{5/4}$
When to apply	When data $\gg$ than key	When data almost sorted	General case

Quicksort

- General purpose sorting.
- Developed by Hoare in 1960.
- Divide and conquer algorithm.

- Idea:
    - Choose one element w arbitrarily, the pivot.
    - Subdivide array in two parts:
        - Set A_1 of elements $\leq w$
        - Set A_2 of elements $> w$
    - If A_1 and A_2 are sorted, END.
    - Else sort A_1 and A_2 recursively.



Quick Sort

- Suppose our array to sort is $[a_1, \ldots, a_N]$
- How do we do the partitioning?
- With indexes!
- Choose element $a_r = w$ as pivot
  - Swap $a_r = w$ and a_N (move pivot to end)
  - Take 2 indexes $l = 1, h = N$
  - Increase l until you find an element $a_l > w$.
  - Decrease h until you find an element $a_h \leq w$.
  - Swap a_l and a_h.
  - Repeat until $l \geq h$.
  - Now swap a_l and w: this brings w at the right place in the field.
- Now do same for arrays $a_1, \ldots, a_l = w$ and $a_{l+1}, \ldots, a_N$

Quick Sort: an example

amheretobesorted ← Original String

amheretọbesorted	Choose pivot (half of field)
amheretdbesorteọ	Swap pivot with last place
a̲mheretdbesorte̲ọ	Set running indices
amher̲etdbesorte̲ọ	count up l till $a_l > w$
amher̲etdbesorte̲ọ	count down h till $a_h \leq w$
amhee̲etdbesortr̲ọ	swap
amheeet̲dbesortr̲ọ	count up l till $a_l > w$
amheeet̲dbeso̲rtrọ	count down h till $a_h \leq w$
amheeeo̲dbest̲rtrọ	swap
amheeeodbes̲t̲rtrọ	count up l till $a_l > w$
amheeeodbes̳trtrọ	count down h: indexes meet
amheeeodbeo̳trtrs	Swap a_l and w
	Check if trivial case
amheeeodbeo trtrs	Separate vectors and continue until sorted

Quicksort speed

Speed:

- Average case needs $2N \log N$ comparisons $\implies \mathcal{O}(N \log N)$
- Worst case: I always choose the pivot so that it is extreme: in this case I need N and not $\log N$ steps $\implies \mathcal{O}(N^2)$

Heap Sort

- A heap is a binary tree, such that each father node is greater or equal to its children nodes.
- This implies the root of the heap is the biggest node.

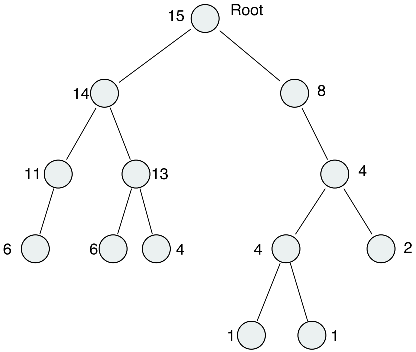


Figure: A heap.

Heap Sort

- How do I build a heap?
- Simple: initialize a tree with one element
- Insert node at free leaf.
- Compare with father, and swap if its smaller
- Push it upwards, until heap condition is satisfied
- Do this for all new elements to sort

Heap Sort

- How do I sort from a heap?
- Simple: pick root
- Take last leaf, move to root
- Push down new root to where it belongs (restore heap condition).
- Pick again from root until tree is empty

Heap Sort

- Speed:
- Algorithm in two parts:
    - Construction of tree: maximum $N \log N$ comparisons
    - Extraction: again $N \log N$ comparisons
- Thus: $2N \log N$ comparisons!! $\Longrightarrow \mathcal{O}(N \log N)$
- Real advantage: by dynamic sorting (constant inserts) resorting real fast due to insertion of single elements and extraction.