Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Introduction to Computer Science
Lecture 04

Version: 2023

Marek Wilkus Ph. D. **http://home.agh.edu.pl/~mwilkus**
Faculty of Metallurgy and Industrial Computer Science
AGH UST Kraków

Sorting algorithms

- Iterative or recursive,
- Comparison-based or non-comparison based,
- Stable or not stable,
- In-place or requiring more memory

Why review sorting algorithms?

- Usefulness - it’s much easier to work on sorted data.
- Sorting algorithms implement many approaches to solve problem.
- Lots of applications:
  - Detecting duplicates,
  - Counting frequency of symbols,
  - Finding subsets,
  - ...and colliding/joining them,
  - Faster searching

Thing we already know...

- Can we use a BST to sort items?

1. Insert items to BST
2. Traverse the BST in order.

- PROBLEMS:
  - If a tree is not balanced, and we have a descending order of adding items, we will get a singly linked list instead → O(n^2).
  - If we use self-balancing trees, we may enhance this to O(n log(n)).
  - We need O(n) of memory space for it.

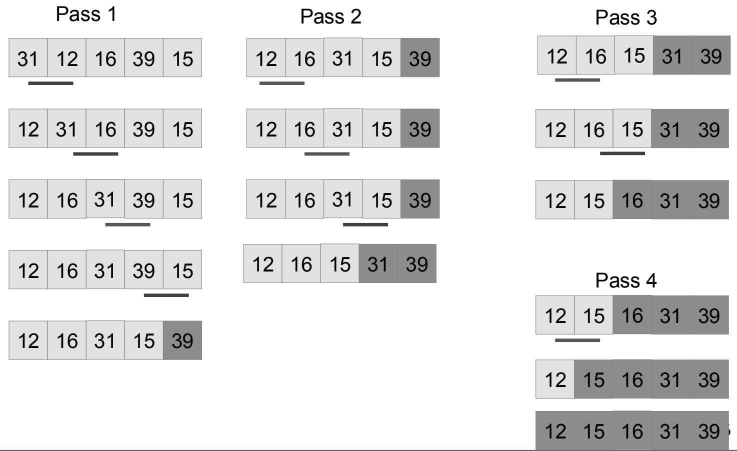
Bubble sort

- We have a n-element unsorted/partially sorted array.
- The procedure:

```
For x [0..n-1]
    1. Compare pair of numbers n and n+1
    2. If out of order → swap them.
  3. Reduce n-1 by 1 and repeat.
```

- Notice how the biggest items „bubble” to the last positions of the array.

Bubble sort



Pass 1
31 12 16 39 15
12 31 16 39 15
12 16 31 39 15
12 16 31 39 15
12 16 31 15 39

Pass 2
12 16 31 15 39
12 16 31 15 39
12 16 31 15 39
12 16 15 31 39

Pass 3
12 16 15 31 39
12 16 15 31 39
12 15 16 31 39

Pass 4
12 15 16 31 39
12 15 16 31 39
12 15 16 31 39

Bubble sort

- Can be implemented using two nested loops.
  - The outer loop spins exactly *n* times.
  - The inner loop's iterations decrease by 1 with each iteration of the outer loop.
- The complexity is O(n^2)

Let's cheat a little

- Notice that the last pass was done without any swapping.
- If the set is sorter earlier, we will just fruitlessly and blindly compare the already sorted array.
- A good way to make things short is to **terminate the algorithm** when we see that the operation is completed.
- The test will look like this:
  - If during the inner loop iteration no swapping has been made → end the algorithm as the set is sorted.
- This will save some time.

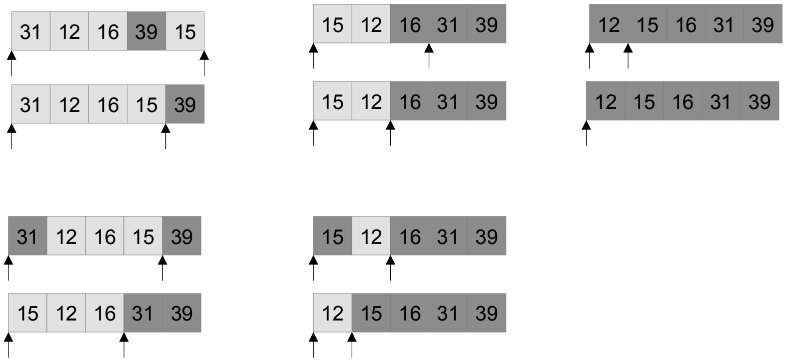
An altered version complexity

- Worst case is the array sorted backwards, our condition will not run and it will be still O(n^2).
- Best case: A fully sorted array. Will end in a single pass - O(n).
- Practical application: We have to add the (small) performance impact of additional variable holding information was there any swapping or not.

Selection Sort

- Find the largest item in 0..n-element array.
- Swap the largest item with the n-th item
- It's in a proper place, so n=n-1 and repeat.

Selection sort



Selection sort

- The most important part is finding the index of the last item from the sub-array [0..n-x]
- The x is the iteration number then.
- So we still use two nested loops:
  - Outer: Thru a whole array.
  - Inner - Thru less and less elements in the array.
- So the complexity is still O(n^2).

Implementation considerations

- If our „swap” is more resource-consuming (we’re not using pointers, we have to re-calculate something) the selection sort will significantly outperform bubble sort.

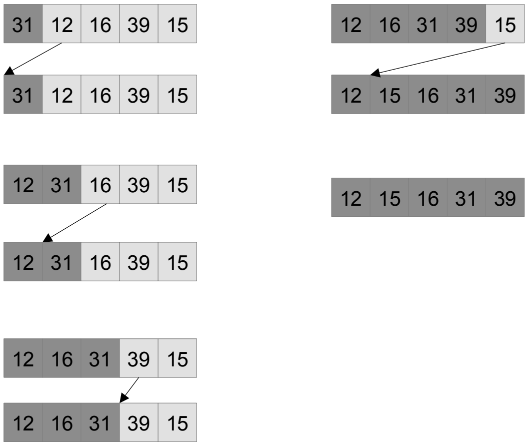
- ...but we cannot use the „cheating” we ued with bubble sort, so we won’t get O(n) in the best case.

- In practical applications, usually selection sort performs a bit better.

Insertion sort

- Start with a single item. It is always sorted.
- Take the next item
- Determine the position in the „sorted” set in which it has to be inserted into.
- Insert the item in its proper position in the „sorted” set.
- Repeat for all unsorted items.

Insertion sort



31 12 16 39 15

31 12 16 39 15

12 31 16 39 15

12 31 16 39 15

12 16 31 39 15

12 16 31 39 15

12 16 31 39 15

12 15 16 31 39

12 15 16 31 39

Insertion sort: Implementation

For every item of index *i* in the n-item array:

```
insertedItem = array[i] ← the item to be inserted.
j = i-1

while (j>0) and (array[j]>insertedItem)
    array[j+1] = array[j] ← shift sorted items to make space for a new one.
    j--

a[j+1]=insertedItem ← insert the item in the correct location.
```

Insertion sort: Complexity

- The outer loop always executes *n*-1 times.
- If the array is already sorted, the inner loop will execute once per item.
- In the worst case, insertion will always occur - the loop will execute always at its full range.

- So the best-case complexity is O(n), and the worst-case O(n^2).

Sorting algorithms stability

- The sorting algorithm is stable when it does not change the relative order of two items with the same value.
- So, with bubble sort, if we swap only if items are in the wrong order, we get the **stable** algorithm. The same with insertion sort.

- Now, the selection sort is **not stable** - starting the largest item lookup from the beginning and inserting it to the end of the sorted set, it will **swap** the order of the largest items with the same value.

Merge sort

- Assume we know how to **merge** two sorted sets into one sorted set:
  - {2, 3, 10} and {5, 17} → {2, 3, 5, 10, 17}
- Can we use it to sort any set?

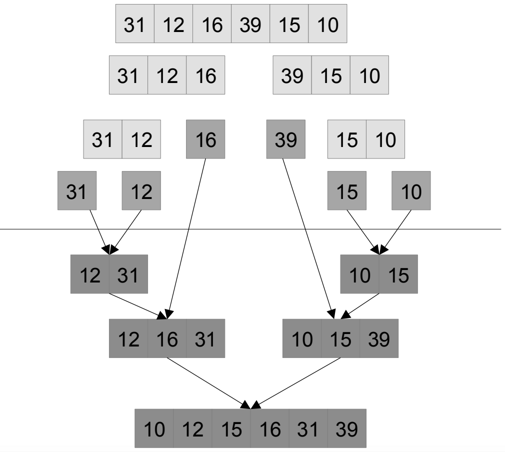
- We can divide sets as we want.
- A set of 1 item is always in order.
- Now we can **merge** 2 1-item sets.
  - Then we **merge** 2 2-item sets,
    - Then we merge 2 4-item sets,
      - Then we merge 2 8-item sets,
        - …
- Until we get a single sorted set.

Divide and conquer method

- First, we divide the problem to the smaller ones.
- Recursively solve the smaller problems.
- Combine the results of the solutions while coming back from the recursion to obtain solution for larger problem.

- So in the Merge sort:
  1. Divide the set to two (equal) halves.
  2. Recursively Merge sort two halves.
  3. Merge two halves of the sorted array.

Merge sort



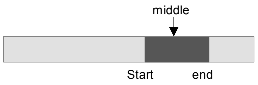
Divide phase (split arrays)

Conquer phase (merge arrays)

Merge - recursive function

```
MergeSort(array, start, end)
   if (start<end)
      middle=start+end/2
      MergeSort(array,start,middle)
      MergeSort(array,middle+1,end)

      Merge(array, start, middle, end)
```



Is there 0 or 1 item?

Merging implementation (1)

```
Merge(array, start, middle, end)
   n=end-start+1
   buffer= n-element array
   left=start
   right=middle+1
   index=0

   while(left<=middle and right<=end)
      if (array[left]<=array[right])
         buffer[index++]=array[left++];
      else
         buffer[index++]=array[right++];
```

This part inserts new items to the buffer, item by item, it choses left-hand or right-hand side to insert from.

Merging implementation (2)

```
while (left<=middle)
   buffer[index++]=array[left++]
while (right<=end)
   buffer[index++]=array[right++]

for every element with index i in buffer
   array[start+i]=buffer[i]
```

All which remains in source sets is copied to the buffer.

Finally, the buffer is copied to the respective place in the array

Complexity of Merge Sort

- Most of work is done in this Merge function.
- For a call of Merge(array, start, middle, end) we have:
  - start-end+1 items
  - At most *number of items*-1 comparisons
  - At most *Number of items* moves to the buffer
  - ...and the same number or moves back to the main array.
  - So at most it's 3* *number of items* - 1 → O(n).
- But we call it **many** times...

When the Merge is called?

- For the single array of *n* items, we don't call the function.
- When it's divided to 2 sets, we have a single call with *n/2* items in each half.
- When it's divided to 4 sets, we have 2 calls, $n/2^2$ items in each half (at most).
- When it's divided to 8 sets, 2^2 calls, $n/2^3$ items in each half.
- The total time complexity of the method is O(n log(n)).

Summing up

- This is considered an optimal comparison-based method.
- Can operate on large data, requiring the buffer.
- It can be proven, that it is stable.
- ...but it is quite problematic to implement at first (recursion!).
- ...and is not an **in-place** method → requires a non-constant size of extra storage.

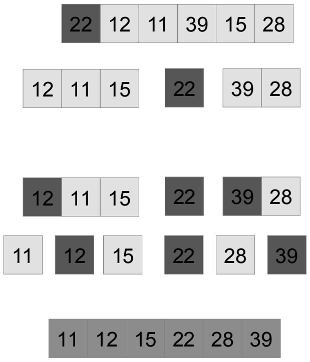
Quick Sort

- In the merge sort, we do most activities in the merge step.
- So in the „divide and conquer" methodology, we do most of activities after the problem has been divided to sub-problems.
- Is it possible to shift the sorting activity into the divide part?

Quick Sort

- Dividing:
  - Choose a specific item **p** known as **pivot**, and divide the set to two subsets:
    - Smaller than **p**,
    - Larger or equal than **p**.
  - Perform the same thing for every part
- ...and nothing left to do after the recursive division.

Quick Sort



Quick Sort

- Notice that the pivot, after executing the sorting round related to it, takes its final position in the given sub-set → it does not participate in further sorting and is in the proper position in this sub-set.
- So the implementation will be:

```
QuickSort(array, start, end)
    if (start<end)
      int pivotIndex = partition(array, start, end)
      QuickSort(array, start, pivotIndex-1)
      QuickSort(array, pivotIndex+1, end)
```

The partitioning function - the most simple implementation

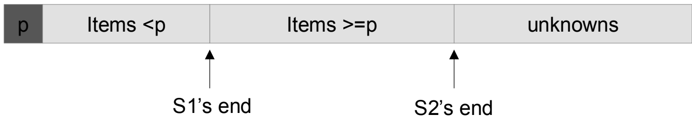
- Assume we will take the 0th item of the array to be partitioned.
- We define two sets (dynamic arrays for example)
  - S1 - in which items are < pivot’s value.
  - S2 - in which items are >= pivot’s value.
- Iteratively we compare all items against the pivot. If they are smaller, they got to S1, else → S2.
- Finally, we return the S1, pivot and S2.

Partitioning - can we optimize it?

- Instead of allocating the memory for two sets, we can **use the array we have.**
- The pivot is array[0] as usual.
- The next parts of the array can be:
  - The first set (S1): <pivot.
  - The second set (S2): >=pivot
  - The last set: Not checked yet.
- We iterate thru items starting from array[0].

Better partitioning approach

- We iterate on the array starting from pivot.
- If the item is >=p, we increment the pointer to the **S2 set**’s end**.**
- Else, we have to increment the S1’s end pointer, but we have an item in ther S2’s range…
  - ...so let’s just swap these items and extend S2’s end pointer too.



The implementation

```
Partition(array, start, end)
    int pivot = array[start]
    int S1end=start
    int S2end=start+1;

    while (S2end<=end)
        if (array[S2end]<pivot) //extend S1
            S1end++
            swap(S2end, S1end)
        S2end++              //we always proceed with S2

    swap(start, S1end) //correct pivot position
    return S1end //return new pivot position
```

Quick sort: Summary

- The algorithm is **not stable**, but **in-place**.
- The best result is obtained if the problem is always divided to two equal halves.
  - Then, depth of recursion is logarithmic,
  - The complexity is O(n log(n)).
- The worst case is when the pivot gets always separated → we get a full S1 and empty S2 or full S1 and empty S1.
  - Then, we get to $O(n^2)$
  - Notice this will happen is we try to sort the already sorted array!
  - It can be fixed with different pivot initialization.

Radix sort

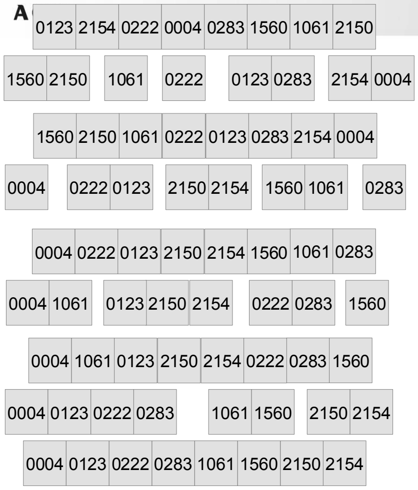
- We have always used sorting algorithms for numbers.
- What if we want to sort strings?
  - Convert strings to ASCII numerals?
  - …?UTF-8 numerals? ← trouble!
- What if we want to develop a sorting method **specifically** for strings?

Radix sort

- We consider each record of data as a string of symbols.

- We group string into sets according to the next symbol in each string.
- Concatenate the sets for the next iteration
- Repeat until sorted.

- Assume we have a constant-length strings.
  - ...but if we stick to numerals, we can zero-pad.



0123 2154 0222 0004 0283 1560 1061 2150

1560 2150   1061   0222   0123 0283   2154 0004 — Grouped by the 4th symbol….

1560 2150 1061 0222 0123 0283 2154 0004 — ...and concatenated back.

0004   0222 0123   2150 2154   1560 1061   0283 — Grouped by the 3rd symbol…

0004 0222 0123 2150 2154 1560 1061 0283 — ...and concatenated back.

0004 1061   0123 2150 2154   0222 0283   1560 — Grouped by 2nd symbol…

0004 1061 0123 2150 2154 0222 0283 1560 — ...and concatenated back.

0004 0123 0222 0283   1061 1560   2150 2154 — Grouped by the 1st symbol…

0004 0123 0222 0283 1061 1560 2150 2154 — ...and it's a sorted set.

A well aligned example from Halim S. - „World of Seven" - https://www.comp.nus.edu.sg/~stevenha/

Radix sort

- How do we group items?
  - We find the first item of the specified characteristics,
  - Move it to the specific set.
  - Go until the end of input data.
  - Repeat for each symbol (or group).

```
So:
i=1;
for every symbol in the record length:
    group the items by i-th item.
    concatenate groups
    i++
```

Radix sort: Grouping and merging

- We can do grouping iteratively:

  Given i = the position we use for grouping.
  Create a set of vectors for every symbol.

```
for every symbol in the array:
    add the symbol to the vector related to it.
```

- For the decimal numbers:

```
digit=0;
for every element of the array
    digit=(element/i) %10
    push(digitsList[digit], element)
```

Merging

- We can just copy the vectors to the array, symbol by symbol:

```
i=0
for every symbol in the alphabet
    while symbol's list not empty
        array[i]=pop(symbol's list)
        i++
```

Radix sort

- We can use any alphabet we want.
- For each iteration we go thru the whole array once to place them to groups, then we concatenate groups to the array.
- So the complexity is O(n).
- Number of iterations: number of symbols in the alphabet.
- So the complexity is O(dn).

- Not in-place, but stable.
- Requires more memory.

Heap sort

- A binary **heap** is a data structure which is a binary tree with the following limitations:
  - All its levels except the last one are completely filled (Shape property).
  - The value stored in each node is greater (or less - depending on implementation) than its children.

- Thanks to shape property, we can **serialize** a BST into the linear array and retrieve it back without any pointers.
- Because it’s not a BST, we can shift elements down without any side effects.

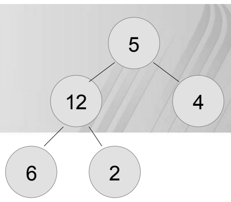
Heap sort

- If we are able to build a heap of the data, we can do the following:

  1. Remove the topmost part of the heap (maximum) and append it to the sorted list part.
  2. Re-create the heap.

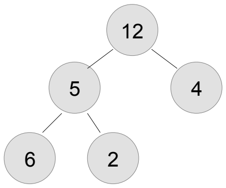
- Because the n-element heap is serialized in the array, we can do the step 1 by moving the maximum to the place right after the end of the heap in the same array - the „heap” part will shrink, the „sorted” part will grow then.

Heap sort



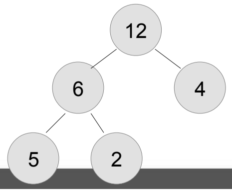
- [5,12,4,6,2]
  - Not a heap

- As 5<12, we swap them

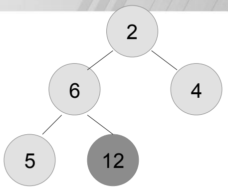


- And as 5<6, we swap 5 with 6.

- We have a heap property.

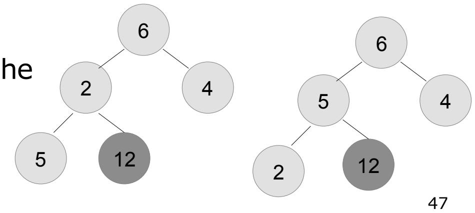


Heap sort



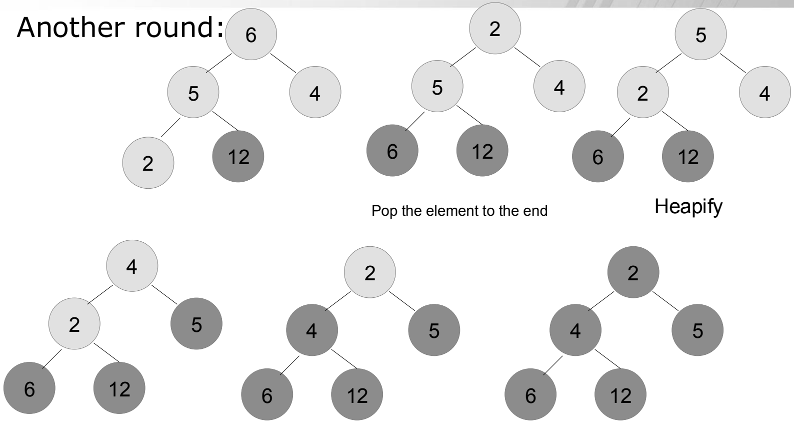
- Remove the root - this is sorted
- We do it by swapping it with the last node and trimming it off from further processing.
-

- Then, we re-create the heap:
  - Swap 6-2
  - Swap 5-6



Heap sort

- Another round:



Pop the element to the end

Heapify

Pop the element

Heap sort: Typical Implementation

```
• HeapSort(array, count)
      start=count/2
      end=count
      while (end>1)
            if (start>0)
                  start--
            else
                  end--
                  swap(array[end],array[0])

            //Moving the smaller elements down
            root=start
            while (2*root+1)<end
                  leaf=2*root+1
                  if (leaf+1<end and a[leaf]<a[leaf+1]
                        leaf++
                  if (a[root]<a[leaf])
                        swap(a[root],a[leaf])
                        root=leaf
                  else
                        break
```

Select the largest item and put it to the end

Restore heap properties.

Heap sort

- Complexity (worst case): O(n log(n))
- Not a stable sort
- In-place algorithm

- In many practical cases a bit slower than Quick sort.

Hybrid algorithms

- How can we get rid of O(n^2) complexity in badly formed input data?
  - Use a different algorithm, not so suitable for other alignments of data…

- Introspective sort variant:
  - If the number of data is relatively small, go with InsertionSort.
  - If the recursion depth of Quick sort is acceptable we can go with Quick sort.
    - ...but we can just partition the array in 2 and go Intro Sort on them.
  - Else, we go with Heap sort.

Estimating the depth of recursion

- Usually 2*$\log_2$n.
- This will „lock" the complexity to O(n log(n)).

- The total complexity of Intro sort will be then O(n log(n))
- ...however, it is not stable.
- ...and more difficult to implement.

A small interruption

- Stack machines
  - A CPU that does not use registers, but (usually one) stack.
  - Execution of operation means:
    - Fetching operation code from the stack,
    - Fetching operands from the stack,
    - Pushing the result to the stack.
    - With 2-operand command, stack is shorter by 1 value or 2 items: instruction and value.

Stack mahines

- Implementations:
  - Separate stack and conventional instructions storage (Symbolics, 1970s).
  - Single stack including instructions and operands (Ferranti, 1965-70s).
  - Multiple stacks, switchable and shortable (Syeika, 1970s, Multiklet project).
- Because it is quite difficult to program these CPUs and typical programming languages compilers don’t generate such code efficiently, they never got popularity.
  - In 1970s, Symbolics LISP-programmable machines were implemented as stack CPUs.

Sorting using stacks

- Having a set of stacks it is possible to implement a sorting algorithm similar to insertion sort.
- It can be implemented using conventional programming techniques, but also with a stack-based processors.
- The „register-as-stack” architecture allow to obtain a very high performance in the implementation.

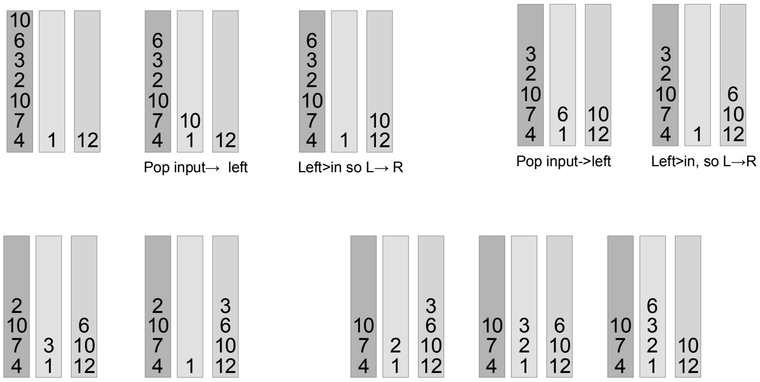
Sorting using stacks

- Rules:
  - No access different than stack-based,
  - No registers, only stacks,
  - The only moving-related operations are push and pop. Popped value must be pushed somewhere.
  - We can compare top values from various stacks without popping.
- Given:
  - Input, unsorted data stack
  - Two working stacks: Left and right.
  - Minimum and maximum element value.

Sorting using stacks

- Initialization:
  - Push the minimum to the left stack,
  - Push the maximum to the right stack.
- Operation:
  - Pop the value from input stack to the left stack.
  - Until top of left stack is not larger than top of input stack:
    - Pop the left stack to the right stack.
  - Until top of the right stack is not smaller than top of input stask:
    - Pop the right stack to the left stack.

Example



Pop input→ left

Left>in so L→ R

Pop input->left

Left>in, so L→R

Pop input->left.

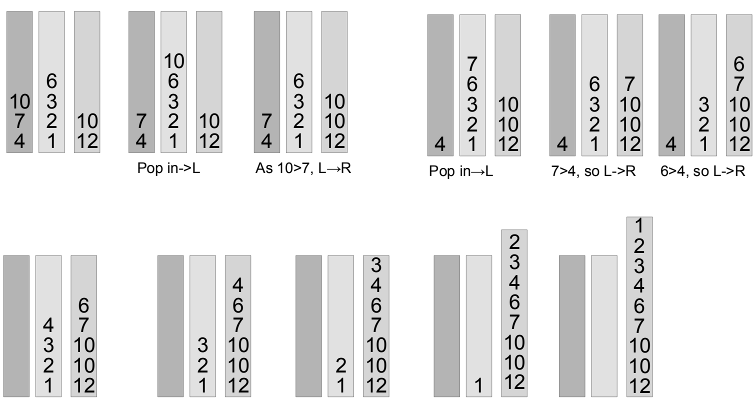
Left>in, so L→R

Pop input->left

Left<in, so R->L

Left<in, so R→L
10!>10, so we don’t pop anymore.

Example (2)



Pop in->L

As 10>7, L→R

Pop in→L

7>4, so L->R

6>4, so L->R

Pop in->L

Transfer L->R

Summing up

- This is an insertion sort.
- Insertion sort can be implemented on stack-like data structures.
- With specific stack machine architectures, it is possible to make it even more efficient.

Mass-solving linear equations

- Any discretized set of differential equations can be described as a system of linear equations.
- The problem is that number of these linear equations may be VERY large.

- Is it possible to solve these equations with a computer?

Linear equations system

- A set of linear equations like:

$$\begin{cases} Ax_1+Bx_2+Cx_3+Dx_4=U \\ Ex_1+Fx_2+Gx_3+Hx_4=V \\ Ix_1+Jx_2+Kx_3+Lx_4=W \\ Mx_1+Nx_2+Ox_3+Px_4=X \end{cases}$$

- Can be described using matrix equation:

$$\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ M & N & O & P \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} U \\ V \\ W \\ X \end{bmatrix}$$

- Sometimes described as:

$$\mathbf{A} \times \mathbf{X} = \mathbf{B}$$

Gauss method

- Allows to obtain solution of such equations system.

- Inspired by the method of solving these systems by eliminating unknowns until we get some x_x=Y, a single unknonwn solved.
  - It can be used to obtain the next equation's unknown,
  - And then, using two solutions, another one…
  - Going this way, we can obtain all unknowns.

Elimination

- If we get a $A_xX=B$ matrix equations (in which X and B are vectors), we can write A|B matrix:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & \dots & a_{2,n} & b_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & \dots & a_{3,n} & b_3 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ a_{n,1} & a_{n,2} & a_{n,3} & \dots & a_{n,n} & b_n \end{bmatrix}$$

Elimination (2)

- Next, we convert all elements under $a_{1,1}$ to 0.
- Then all elements under $a_{2,2}$,
- Then, $a_{3,3}$ etc.

  (the first row remains the same)
- So we can use a single solution to expand it to the rest.

Elimination (3)

- Elimination of column 2 (so only $a_{1,1}$ remains):
  - For all elements of the row *i* (2..n) we add the next elements of the row 1 multiplied by $-1*(a_{i,1}/a_{1,1})$.
  - Notice that for $a_{2,1}$, we will get:
    - $a_{2,1}-(a_{2,1}/a_{1,1})*a_{1,1} = a_{2,1}-a_{2,1} = \mathbf{0}$
    - ...and that's what we want!
  - For a2,2 we will get: $a_{2,2}-(a_{2,1}/a_{1,1})*a_{1,2}$
  - For a2,n we will get: $a_{2,n}-(a_{2,1}/a_{1,1})*a_{1,n}$
  - For b column: $b_2-(a_{2,1}/a_{1,1})b_1$

Elimination

- We process it until we will get the matrix **triangular**:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,n} & b_1 \\ 0 & a'_{2,2} & a'_{2,3} & \dots & a'_{2,n} & b'_2 \\ 0 & 0 & a'_{3,3} & \dots & a'_{3,n} & b'_3 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & a'_{n,n} & b'_n \end{bmatrix}$$

Unpack it back

- We will then unpack the matrix to something that looks more like an equation system:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,n} \\ 0 & a'_{2,2} & a'_{2,3} & \dots & a'_{2,n} \\ 0 & 0 & a'_{3,3} & \dots & a'_{3,n} \\ 0 & 0 & 0 & \dots & a'_{4,n} \\ . & . & . & & \\ 0 & 0 & 0 & \dots & a'_{n,n} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ \\ b'_n \end{bmatrix}$$

Find the unknowns

- Formula for the unknown x_i:

  For $i=n-1, n-2, \dots 1$:

  $x_i = (b'_i-a'_{i,n}x_n-\dots a'_{1,i+1}x_{i+1})/a'_{i,i}$

- So:

  $x_n = b'_n/a'_{n,n}$

- Knowing that subsequent '-operations are recursive, it can be implemented recursively.

Pseudocode

Stage 1: Gaussian elimination

```
For i=1,2,...n-1
   For j=1,2,...n
      If AB[i,i]==0
         return 2        //Error! We are going to divide by 0!
      multiplier = AB[j,i]/AB[i,i]
      For k=i+1..n+1
         AB[j,k]=B[j,k]+multiplier*AB[i,k]
```

WARNING!
==0 is assumed to have some tolerance!

Pseudocode

Stage 2: Obtaining unknown values

```
For i=n, n-1, n-2 … 1
   s=AB[i, n+1]
   For j=n, n-1, n-2 … i+1
      s=s-AB[i,j]*x[j]
   if AB[i,i]==0
      return 2      //ERROR - we are going to divide by 0
   X[i]=s/AB[i,i]
```

- Unknowns are stored in X vector

Crout's enhancement

- Presence of any 0 in the dagonal of the matrix, or introduction of such 0, will cause the algorithm to divide by 0.

- Because A+B==B+A, we can swap columns in the array as we wish.
- Implementation of column swap can be done just using pointer swap.
- Can we swap columns to not get zeros in the diagonal, or better, get them in part we want?

- Wa search for the element with biggest absolute value.
- Now, we swap columns: Column with this element with column with the part of the diagonal.
- It can minimize the division by zero errors.

- Instead of pointers, a **lookup table** can be used.

Even better version

- In practical applications, frequently we have this A×x=b problem with various b values for the same, or similar, A values.
  - Like the same discretized continuous problems, for the same equations, for different points in the medium.

- We can save the eliminated A-values and use them again.
  - ...but there is a b-vector involved - the decomposition must be different.

A = LU

- We describe the converted A matrix as a matrix product of upper and lower triangular matrices:

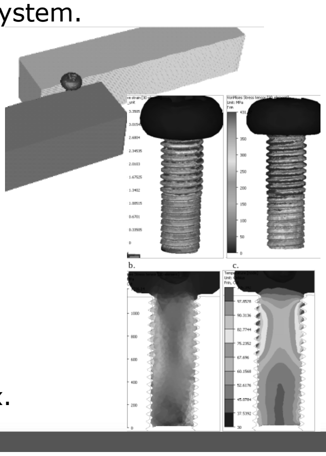
A=LU

- Once we decompose A to LU, we can save them and substitute with any values of B we want.
- The complexity remains as in Gaussian elimination.

Why do we need this?

- A lot of physical, mechanical, structural problems can be converted into a linear equations system.
- There are even more efficient methods to do this.
- If we discretize the continuous media and interpolate in between, we can solve non-linear problems using linear equations!

- ...however, the typical method for an e.g. computer simulations has millions of columns and rows.
  - ...fortunately it is a sparse matrix.



Graph algorithms

- A **graph** is a data structure which consists of **vertices** and **edges** linking them.
- Both vertices and egdes may hold additional information.
- The g**raph order** is a number of vertices in the graph.
- The **graph size** is a number of edges in graph.
- A **null graph** consists vertices, but no edges.

- Graphs are used in discrete mathematics, geometry/topology, and, in application, in engineering.

Properties of graph elements

- Graph may allow a **multi-edge** connection, it means that two vertices may be connected by more than one relation.
- A **loop** is a connection to itself.

- The edge may be **uni- or bidirectional.**
- The graph is **planar** if we can daw it without crossing the edges.
  - Finding planar graph of an existing graph is an important problem in design software algorithms.

Properties of graph elements

- Graph's edges may have their values, called **weights**.
- A **path** is a series of vertex traversals from one vertex to another, usually thru other vertices.
  - Finding the shortest path is an important problem in function optimization and logistics.

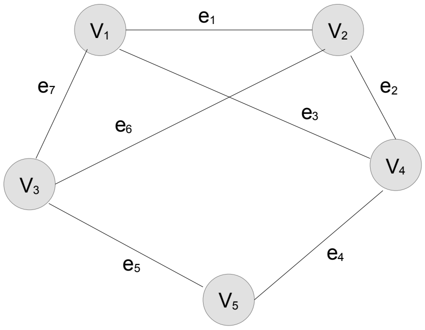
- A **simple graph** has no multi-edges or loops.

Paths and cycles

- A **Hamiltonian path** is a path which goes thru all vertices of the graph.
- A **Hamiltionian cycle** is similarly, a closed path. **The simple cycle must cross each vertex only once**. Some edges may not be used.

- An **Eulerian path** goes thru all edges.
- In the **Eulerian cycle** the path must be closed.

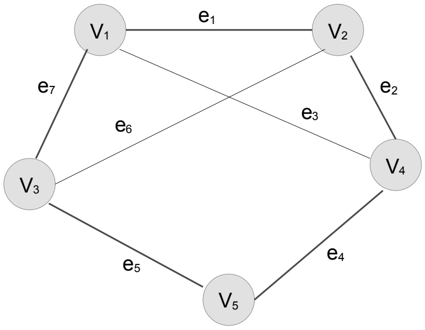
Graph example



A **non-directed, simple** but **non-planar** graph.

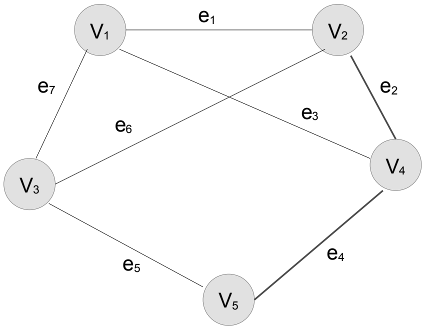
Not a **complete** graph as V_5 is not directly connected to e.g. V_1 or V_2.

Graph example



A Hamilton cycle

Graph example



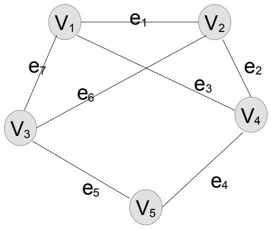
A V_5 to V_2 **path**.

How graph can be stored?

- Structures and pointers
  - PROBLEM: If we may have any munber of edges from a vertex, we need dynamic data structure to hold pointers.
  - PROBLEM: Lack of general overview of the graph.
  - PROBLEM: Algorithms which look for specific vertex will have to traverse it almost blindly.

Better way to store graphs?

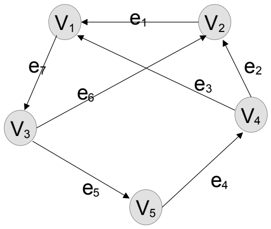
- Adjacency matrix:
  - For **n** vertices we create **n**x**n matrix** of binary values.
  - 1 if there is a connection between column- and row-related element.
  - We can store **directed** and non-directed graphs.



	V_1	V_2	V_3	V_4	V_5
V_1	0	1	1	1	0
V_2	1	0	1	1	0
V_3	1	1	0	0	1
V_4	1	1	0	0	1
V_5	0	0	1	1	0

Better way to store graphs?

- Adjacency matrix and directed graphs:
  - Rows contain starting points
  - Columns: End of edge.



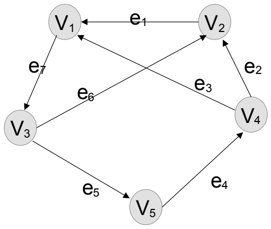
	V_1	V_2	V_3	V_4	V_5
V_1	0	0	1	0	0
V_2	1	0	0	0	0
V_3	0	1	0	0	1
V_4	1	1	0	0	0
V_5	0	0	0	1	0

Using the adjacency matrix

- A degree of the vertex, for a non-directed graph, is a count of 1s in the column or row related to this vertex.
- For a directed graph, we obtain, by counting in columns or rows, degree of „inputs" or „outputs" of the vertex.
- In a directed graph, if [x,y]==[y,x]==1, then these two vertices have two links (or one bi-directional, if we allow it).

Incidence matrix

- Each row is a vertex,
- Each column is an edge,
- Each value is a relation:
  - 0, if there is no relation between vertex and edge,
  - 1 if a vertex is a start of the edge,
  - -1 if a vertex is the end of the edge.



	e_1	e_2	e_3	e_4	e_5	e_6	e_7
V_1	-1	0	-1	0	0	0	1
V_2	1	-1	0	0	0	-1	0
V_3	0	0	0	0	1	1	-1
V_4	0	1	1	-1	0	0	0
V_5	0	0	0	1	-1	0	0

Properties of incidence matrix

- Much easier to add weights than in neighbourhood matrix.
- Each column must have one -1 and one 1 (integrity check).
- Number of 1s and -1s in rows → number of edges in and out.
- Finding neighbours is harder as we have to seek thru an entire row.
- All zeros in a row → insulated vertex.

	e_1	e_2	e_3	e_4	e_5	e_6	e_7
V_1	-1	0	-1	0	0	0	1
V_2	1	-1	0	0	0	-1	0
V_3	0	0	0	0	1	1	-1
V_4	0	1	1	-1	0	0	0
V_5	0	0	0	1	-1	0	0