Elementary maths for GMT

Algorithm analysis

Part II

Algorithms, Big-Oh and Big-Omega

- An algorithm has a $O(\cdots)$ and $\mathbf{\Omega}(\cdots)$ running time
- By default, we mean the *worst case* running time
- A worst case $O(\cdots)$ running time is a statement about *all* possible inputs
- A worst case $\mathbf{\Omega}(\cdots)$ running time is a statement about *one* input

Algorithms, Big-Oh and Big-Omega

- Consider the following `BubbleSort` algorithm

Algorithm ***BubbleSort(X)***
Input array X of n integers
Output the sorted version in array X

for $i \leftarrow 1$ **to** $n - 1$ **do**
$j \leftarrow i$
while ($j > 0$) **and** $X[j] < X[j\text{-}1]$ **do**
swap $X[j]$ **and** $X[j\text{-}1]$
$j \leftarrow j - 1$
return X

Algorithms, Big-Oh and Big-Omega

- If ***X*** is already sorted, then `BubbleSort` runs in $O(n)$ time
- If ***X*** is sorted in reverse order, then `BubbleSort` runs in $O(n^2)$ time
- If ***X*** is in any other permutation, the running time is somewhere in between

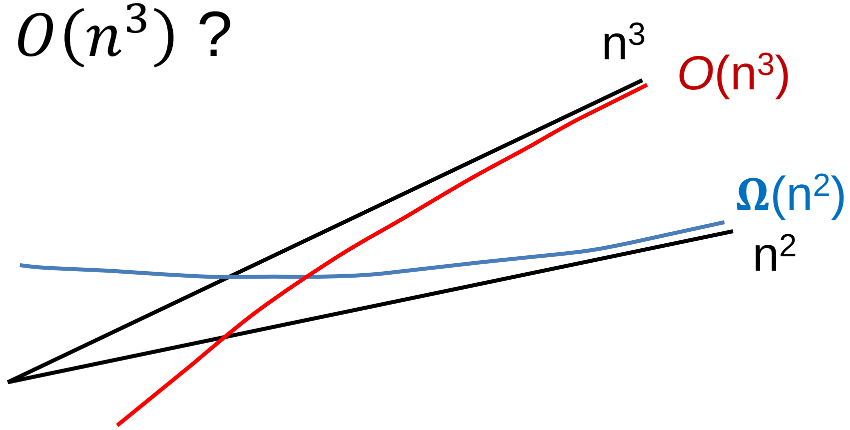
➢ The worst case running time is $O(n^2)$

Algorithms, Big-Oh and Big-Omega

- If ***X*** is already sorted, then `BubbleSort` runs in $\mathbf{\Omega}(n)$ time
  - we can claim $\mathbf{\Omega}(n)$ running time in the worst case
- If ***X*** is sorted in reverse order, then `BubbleSort` runs in $\mathbf{\Omega}(n^2)$ time
  - we can claim $\mathbf{\Omega}(n^2)$ running time in the worst case (which is a stronger claim)
- Since the running time is also $O(n^2)$ in the worst case, we cannot find an even worse input

Algorithms, Big-Oh and Big-Omega

- Since the worst case running time is $\mathbf{\Omega}(n^2)$ and $O(n^2)$, the running time is also $\Theta(n^2)$ in the worst case
- The worst case running time bound is **tight** if the upper bound and lower bound match
- Is it possible that an algorithm has a worst case running time of $\mathbf{\Omega}(n^2)$ and $O(n^3)$?

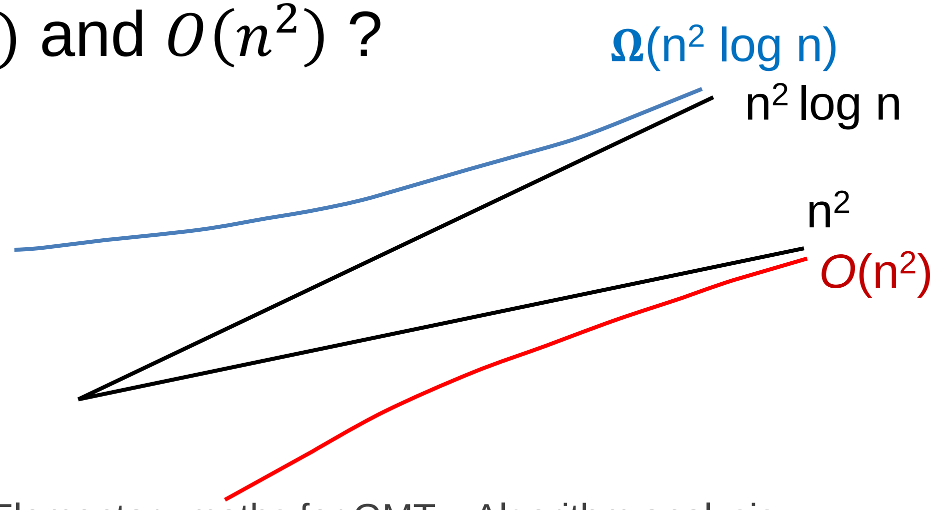


Algorithms, Big-Oh and Big-Omega

- Since the worst case running time is $\mathbf{\Omega}(n^2)$ and $O(n^2)$, the running time is also $\Theta(n^2)$ in the worst case

- The worst case running time bound is **tight** if the upper bound and lower bound match

- Is it possible that an algorithm has a worst case running time of $\mathbf{\Omega}(n^2 \log n)$ and $O(n^2)$?



Basic algorithm problems

- The problem of **sorting** a set of numbers is perhaps the most fundamental algorithmic problem
- `InsertionSort` and `BubbleSort` are simple incremental algorithms that take $\Theta(n^2)$ time in the worst case
- `MergeSort` and `QuickSort` are based on a divide-and-conquer approach and take $\Theta(n \log n)$ time in the worst case
- `CountingSort` takes $\Theta(n)$ time but only works for integers that are not too large
- Is it possible that any sorting algorithm is even faster than $\Theta(n)$ time?

Basic algorithm problems

- The problem of **storing** a set of numbers for efficient **searching** is the most fundamental **data structuring** problem
- A sorted array allows for **binary search**, which take $\Theta(\log n)$ time (binary search is a search algorithm for a single number in a sorted set)
- In an unsorted array, searching cannot be faster than $\Theta(n)$ time
- **Hash tables** are specifically organized arrays that allow searching in $\Theta(1)$ time in practice, but not as a worst case bound

Recall: important functions

Function	Time	Usage in
Constant	$O(1)$	initialization of a variable
Logarithmic	$O(\log n)$	**searching** in a sorted set
Linear	$O(n)$	A full **scan** over the input
N-Log-N	$O(n \log n)$	**sorting** a set
Quadratic	$O(n^2)$	nested loops
Cubic	$O(n^3)$	one deeper nesting
Exponential	$O(2^n)$	**all subsets** of a set
Factorial	$O(n!)$	**all ordering** of a set

Different steps in an algorithm

- Consider the problem: given a set of *n* numbers, are any two equal?
  - Example: 4, 6, 14, 3, 7, 97, 56, -4, 89, 34, 8, 14, -23, 88
- Solution 1 – The intuitive way: consider all pairs of numbers and test each pair
  - Result in a $O(n^2)$ algorithm (nested loops)
- Solution 2 – The sort&search approach: sort the numbers with `MergeSort` or `QuickSort` (step 1) and then scan (step 2) to see if two *adjacent* numbers are equal
  - Step 1 takes $O(n \log n)$ time and step 2 takes $O(n)$ time
  - In total $O(n \log n) + O(n) = O(n \log n + n) = O(n \log n)$ time

Different steps in an algorithm

- An algorithm has different steps if it has subtasks and each subtask is completely finished before the next one begins
- We analyze each subtask separately and add up their running times
- With Big-Oh notation and removal of constants and lower-order terms, this implies that the most time expensive subtask determines the efficiency of the whole algorithm

Different steps in an algorithm

- Compare the two following algorithms

```
Algorithm Loops1(X)
    Input array X of n integers
    Output irrelevant

    for i ← 1 to n do
        some computations
        for j ← 1 to n do
            some computations

    return something
```

```
Algorithm Loops2(X)
    Input array X of n integers
    Output irrelevant

    for i ← 1 to n do
        some computations
    for j ← 1 to n do
        some computations

    return something
```

- What is their running time?

More example problems

- Given a set of *n* numbers, can we split them in two subsets with the same summed value?
  - set: -18, 4, 22, 14, 2, 7, 97, 56, -6, 88, 34, 9, 17, -23, 69
  - total sum is 372, half is 186
  - One solution: -23,2,22,88,97 and -18,-6,4,7,9,14,17,34,56,69

Another nested-loops example

- Analyze the following algorithm

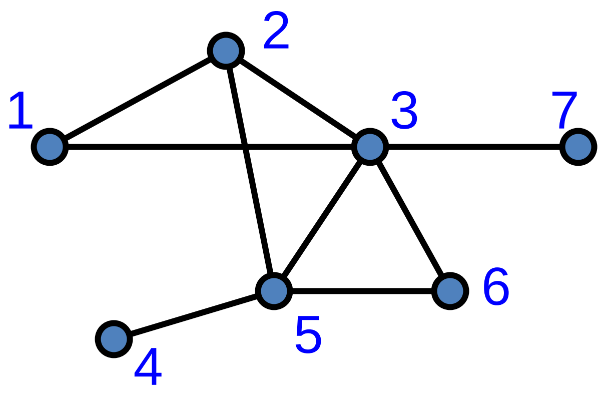
```
Algorithm SumOccurs(X, m)
    Input array X of n integers and an integer m
    Output true if X[i] + X[j] = m for some i != j

    MergeSort(X)
    i ← 0
    j ← n - 1
    while (i < j) do
        while (X[i] + X[j] > m) do j ← j - 1
        if (X[i] + X[j] = m) then return true
        i ← i +1
    return false
```

Another nested-loops example

- Nested loops (both over the input size) do not always give a worst case quadratic running time
- When not, you need a different argument to bound the number of times the inner loop is executed
- This involves understanding what the algorithm precisely does
  - If you designed the algorithm, you (should) understand what it does
  - Otherwise, applying the algorithm to some example input helps to understand how the algorithm works

Graphs and representations

- A graph $G = (V, E)$ consists of a set V of vertices and a set E of edges
- Abstractly speaking, vertices are elements and edges are pairs of elements
- One can draw a graph by giving coordinates to the vertices, but any graph exists without coordinates
- Example
  - $V = \{1, 2, 3, 4, 5, 6, 7\}$
  - $E = \{(1,2), (1,3), (2,3), (2,5), (3,5), (4,5), (3,6), (3,7), (5,6)\}$

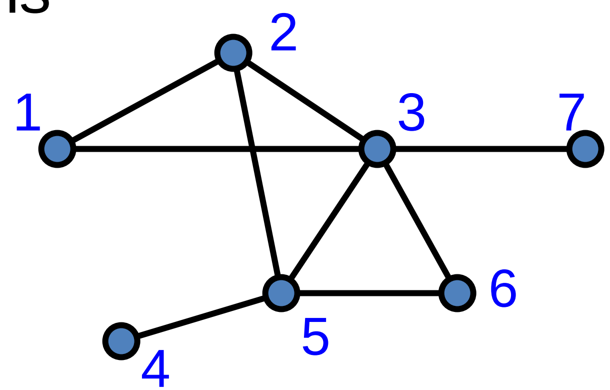


Graphs and representations

- The (input) size of a graph is expressed as the number of vertices and the number of edges: $|V| = n$ and $|E| = m$
- Question: what is the minimum and maximum number of edges a graph with *n* vertices can have?

Graphs and representations

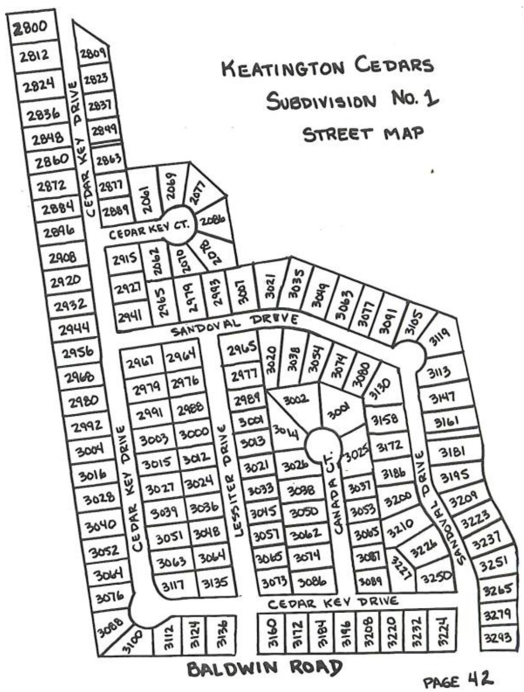
- A graph $G = (V, E)$ is planar if it can be drawn in the plane without any edge-edge intersections
  - Is this graph planar?

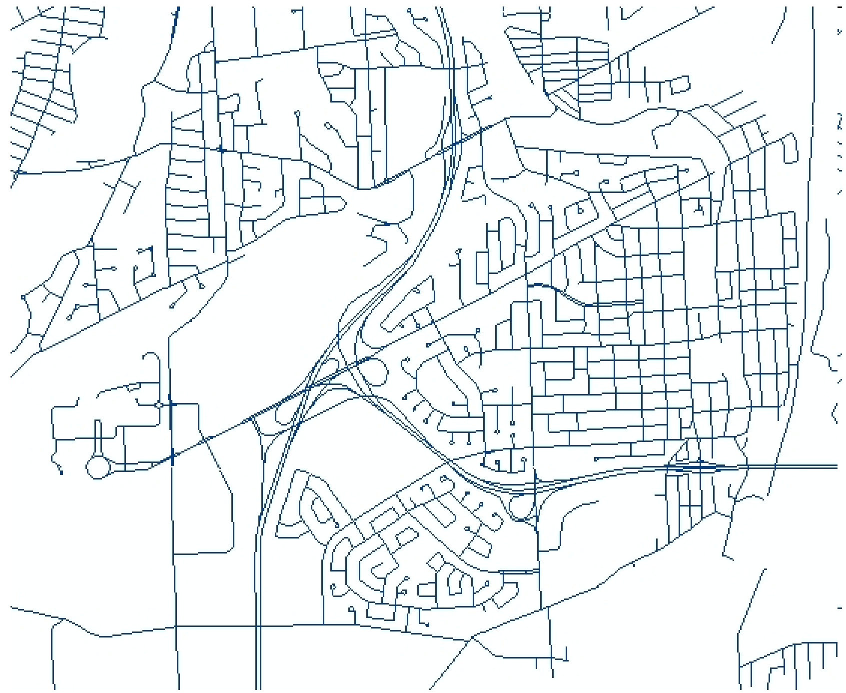


- For planar graphs, it is known that $m \leq 3n - 5$
  - In other words, the number of edges of a planar graph with *n* vertices is $O(n)$
  - Big-Oh notation is not used only for running time statements

Graphs and representations

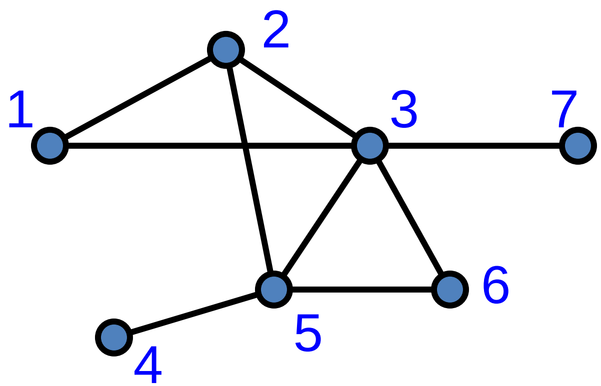
- Subdivisions of the plane can be represented with graphs, if we give coordinates to each vertex
- Road networks are also graphs that have vertices with coordinates





Graphs and representations

- A common representation of a graph is the **adjacency matrix**, a *n x n* matrix of zeroes and ones with a one at *(i,j)* if and only if *(i,j)* is an edge in *E*



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

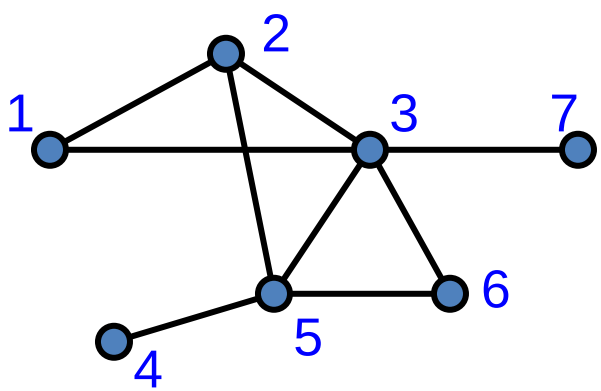
  - $V = \{1, 2, 3, 4, 5, 6, 7\}$
  - $E = \{(1,2), (1,3), (2,3), (2,5), (3,5), (4,5), (3,6), (3,7), (5,6)\}$

Graphs and representations

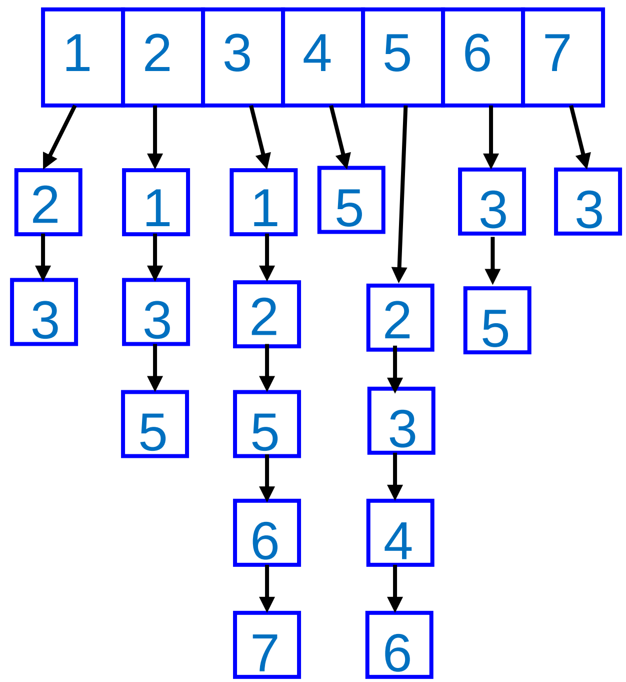
- Some questions
  - Suppose that a graph G with n vertices and m edges is given. How much storage space does the adjacency matrix representation of G need? What if G is planar?
  - Can we use Big-Oh notation to state this?
  - Is the adjacency matrix representation suitable for representing planar graphs?

Graphs and representations

- A different common representation for graphs is the **adjacency list** representation
- It consists of an array $A[1 \cdots n]$, with one entry for each vertex, with access to a list of neighbors of that vertex



  - $V = \{1, 2, 3, 4, 5, 6, 7\}$
  - $E = \{(1,2), (1,3), (2,3), (2,5), (3,5), (4,5), (3,6), (3,7), (5,6)\}$



Graphs and representations

- What are the storage requirements of an adjacency list representation of a graph *G* with *n* vertices and m edges?

  – $O(n+m)$

- Do we really need the *n* and the *m* in the storage bound (for example, would $O(n)$ or $O(m)$ be correct)?

  – We really need both

    - a graph with *n* vertices and $\frac{n(n-1)}{2}$ edges (all possible edges) needs $\Theta(m)=\Theta(n^2)$ storage, and this is not $O(n)$
    - a graph with *n* vertices but no edges needs $\Theta(n)$ storage, and this is not $O(m)$ since $m=0$

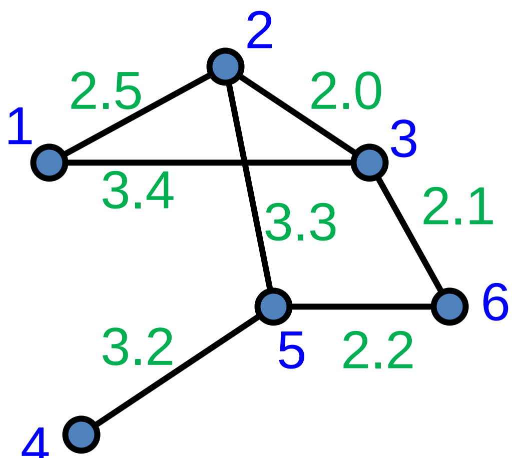
Graphs and representations

- What are the advantages and the disadvantages of the **adjacency matrix** and **adjacency list** representations?

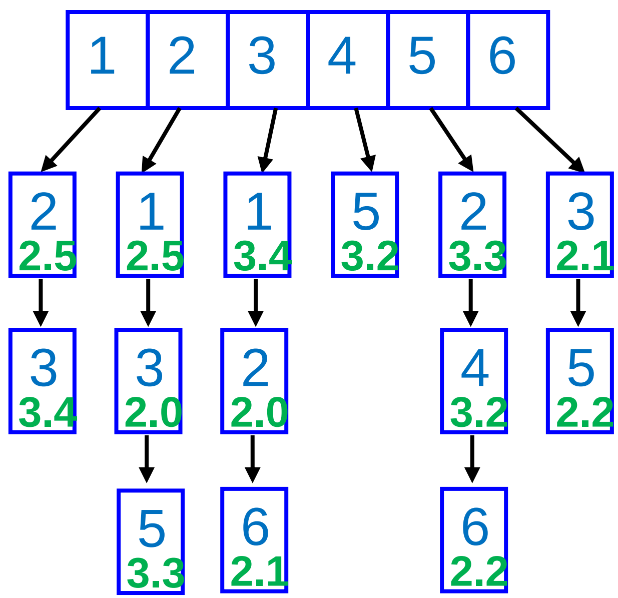
Graphs and representations

- Graphs often have **weighted edges**
  - The weight may represent the distance between the incident vertices, the travel time, the capacity, the cost ...
- In an adjacency matrix, we can simply store the weight of an edge *(i,j)* in the matrix (if no edge is present, we need to use a special value that does not occur as a weight)
- In an adjacency list, we store twice the weight of an edge
  - with *j* in the list of *i*
  - with *i* in the list of *j*

Graphs and representations



$$\begin{pmatrix} - & 2.5 & 3.4 & - & - & - \\ 2.5 & - & 2.0 & - & 3.3 & - \\ 3.4 & 2.0 & - & - & - & 2.1 \\ - & - & - & - & 3.2 & - \\ - & 3.3 & - & 3.2 & - & 2.2 \\ - & - & 2.1 & - & 2.2 & - \end{pmatrix}$$



Graphs and representations

- The most important algorithmic problem on (weighted) graphs is computing shortest paths (sequences of edges with minimum sum of weights)
- A famous algorithm is Dijkstra's algorithm (1959), where a shortest path between two given vertices in a given weighted graph is computed in $O(n + m \log m)$ time

- What graph representation is assumed when we state this time bound?

Some graph problems

- Given a graph
  - decide if a tour exists that visits every edge exactly once
  - decide if a tour exists that visit every vertex exactly once
  - find the largest completely interconnected sub-graph
  - find the largest non-connected sub-graph
  - determine the minimum number of colors to color the vertices so that neighbors have different colors
- Given a planar graph, determine if the vertices can be colored using two/three/four colors so that neighbors have different colors

A geometric problem

- Assume that a computer (model) can do additions, subtractions, multiplications, divisions and memory reads and writes in constant time each
- Given a simple polygon with n vertices, is it algorithmically easier to compute its area or its perimeter?