Stable Marriage Problem

Debarka Sengupta

Background

1.1 Hospitals/Residents Problem

- Each college can admit multiple students, One student can study in only one college (One to Many)
- Cropped up in the mid-40's
- Medical students, when they received offers from hospitals, they used to wait in case a better offer would present itself from a hospital more to their liking.
- The situation would end up in:
  - unhappy students who accepted their first offers
  - unhappy hospitals when students did not keep their earlier commitments.

1.2 Stable Allocation Problem

- Each server can be matched to multiple clients, Each client’s request may be processed at multiple server (Many to Many)
- Key criteria is performance enhancement which may be based on assignment of servers at a closer geographical location.
- The problem is similar to TA assignment to courses in universities.

1.3 Kidney Allocation Problem

- Many donors may be a match for a kidney transplant recipient (Many to One)
- The donors may be deceased or living
- Many willing donors for a specific recipient may not be a match
- There are many criteria in an allocation:
  - matching (compatible immune system, blood type & tissue type)
  - time of transplant (in case of deceased donor)
- Main goal/challenge: thicken the kidney exchange market to enable as many matches as possible

Stable Marriage Problem

2.1 Problem Statement

- The Stable Marriage Problem (SMP) states that:
  - given *n* men and *n* women (where each person has ranked all members of the opposite sex in order of preference)
  - marries the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners.

- If there are no such people, all the marriages are “stable”

2.2 Analysis of Stable Marriage Problem - 1

- Problem of finding a stable matching between two equally sized sets of elements given an ordering of preferences for each element.
- A matching is a mapping from the elements of one set to the elements of the other set.
- A matching is not stable if:
  - There is an element A of the first matched set which prefers some given element B of the second matched set over the element to which A is already matched, and
  - B also prefers A over the element to which B is already matched.

2.2 Analysis of Stable Marriage Problem - 2

- One man will matched to one woman (One to One)
- Complete Bipartite Graph with *2n* vertices:
  - *n* vertices representing men
  - *n* vertices representing women
- In 1962, David Gale and Lloyd Shapley proved that, for any equal number of men and women, it is always possible to solve the SMP and make all marriages stable.

2.3 Gale Shapley Algorithm (GS)

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Example matching 1

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Is this a stable matching?

Example matching 1

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

No!
Albert and Emily form a **blocking pair**

1. Everyone is unmatched

2. While some man *m* is unmatched:

- *w* := *m*'s most-preferred woman to whom he has not proposed yet
- If *w* is also unmatched:
  - *w* and *m* are engaged
- Else if *w* prefers *m* to her current match *m*'
  - *w* and *m* are engaged, *m*' is unmatched
- Else: *w* rejects *m*

3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

1. Everyone is unmatched
2. While some man *m* is unmatched:
   - *w* := *m*'s most-preferred woman to whom he has not proposed yet
   - If *w* is also unmatched:
     - *w* and *m* are engaged
   - Else if *w* prefers *m* to her current match *m*'
     - *w* and *m* are engaged, *m*' is unmatched
   - Else: *w* rejects *m*
3. Return matched pairs

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Men's Preference

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Women's Preference

Analysis

Claim 1 - GS terminates in polynomial time (at most n^2 iterations of the outer loop)

Proof:

- Each iteration, one man proposes to someone to whom he has never proposed before
- *n* men, *n* women -> $n \times n$ possible events

(Can tighten a bit to $n(n - 1) + 1$ iterations.)

Claim 2 - GS results in a perfect matching

Proof by contradiction:

- Suppose by way of contradiction that *m* is unmatched at termination
- *n* men, *n* women -> *w* is unmatched, too
- Once a woman is matched, she is never unmatched; she only swaps partners. Thus, nobody proposed to *w*
- *m* proposed to everyone (by def. of GS)

Claim 3 - GS results in a stable matching, i.e., there are no blocking pairs

Proof by contradiction (1):

- Assume *m* and *w* form a blocking pair

Case #1: *m* never proposed to *w*

- GS: men propose in order of preferences
- *m* prefers current partner *w'* > *w*
- -> *m* and *w* are not blocking

Claim 3 contd. - GS results in a stable matching, i.e., there are no blocking pairs

Proof by contradiction (2):

Case #2: *m* proposed to *w*

- *w* rejected *m* at some point
- GS: women only reject for better partners
- *w* prefers current partner *m’* > *m*
- -> *m* and *w* are not blocking

Case #1 and #2 exhaust space.

Man Optimality/Pessimality

- Let *S* be the set of stable matchings
- *m* is a **valid partner** of *w* if there exists some stable matching *S* in *S* where they are paired
- A matching is **man optimal** if each man receives his *best* valid partner
  - Is this a perfect matching? Stable?
- A matching is **man pessimal** if each man receives his *worst* valid partner

Claim 4 - GS with the man proposing – results in a man-optimal matching

Proof by contradiction (1):

- Men propose in order -> at least one man was rejected by a valid partner
- Let *m* and *w* be the first such reject in *S*
- This happens because *w* chose some *m'* > *m*
- Let *S'* be a stable matching with *m, w* paired (*S'* exists by def. of valid)

Claim 4 contd. - GS with the man proposing – results in a man- optimal matching

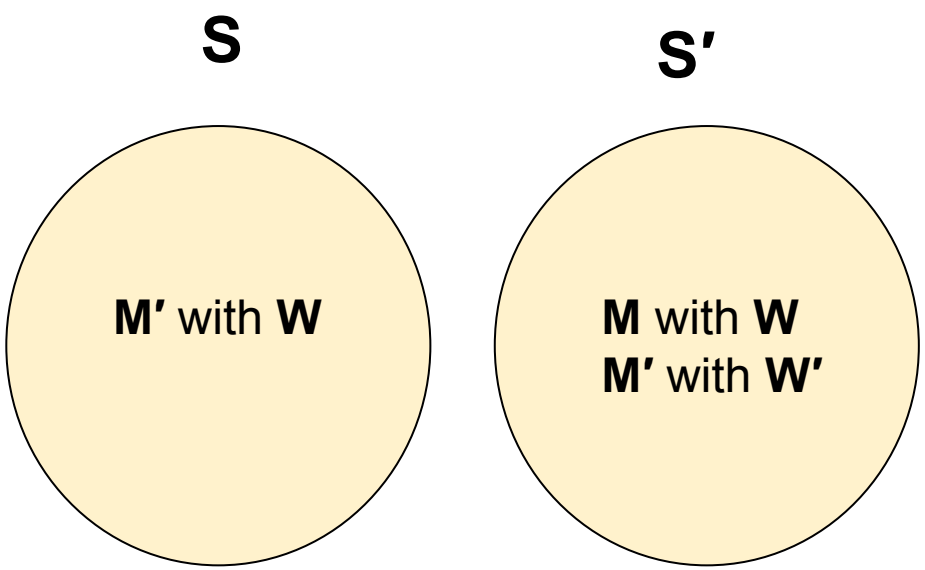
Proof by contradiction (2):

- Let *w'* be partner of *m'* in *S'*
- *m'* was not rejected by valid woman in *S* before *m* was rejected by *w* (by assump.)

  -> *m'* prefers *w* to *w'*
- Now *w* prefers *m'* over *m*, her partner in *S'*

  -> *m'* and *w* form a blocking pair in *S'*

So what's going on here?



1. For W → M′ > M (W rejects M in S)
2. For M′ → W > W′ (As M was the first to be rejected in S)
3. If the above are true, in S′, M′ and W creates blocking pair

Claim 5 - Order in which men propose does not impact the solution

- *Is the above claim just?*

Claim 6 - GS with the man proposing – results in a woman- pessimal matching

Claim 6 - GS with the man proposing – results in a woman- pessimal matching

- *m* and *w* matched in *S, m* is not worst valid
- -> exists stable *S'* (Non GS) with *w* paired to *m'* < *m*
- Let *w'* be partner of *m* in *S'*
- *m* prefers *w* over *w'* (by man-optimality)
- -> *m* and *w* form blocking pair in *S'*

Last but not least

Gale and Shapley proved that it is always possible to find a matching that makes all marriages stable, and provided a quadratic time algorithm which can be used to find one of two extreme stable marriages, the so-called male optimal or female optimal solutions.

For *n* men, *n* women Exponentially large number of stable matchings are possible. Giving a tight upper bound is an open challenge.

4. Miscellaneous

Nobel Prize

- Lloyd Shapley and Alvin E. Roth won the Nobel Prize in 2012 for their work on matching theory, including the kidney donor matching problem.

5. References

1. The Stable Marriage Problem – Optimizing Different Criteria using Genetic Algorithms
https://pdfs.semanticscholar.org/4d33/ecab3803df26695bc1d285ad17ca55192b76.pdf

2. Matching Theory: Kidney Allocation by Kyle Luong (Meds 2016)
http://www.uwomj.com/wp-content/uploads/2013/10/v82no1_6.pdf

3. Matching Kidney Donors with Those Who Need Them & Other Explorations in Economics
https://www.nap.edu/read/23508/pdf/frtr_annotated_kidney.pdf

4. CS364A: Algorithmic Game Theory Lecture #10: Kidney Exchange and Stable Matching*
Tim Roughgarden
https://theory.stanford.edu/~tim/f13/l/l10.pdf

5. Wikipedia page on Stable Marriage Problem
https://en.wikipedia.org/wiki/Stable_marriage_problem

6. Stable Matching John P. Dickerson, CMU
http://www.cs.cmu.edu/~arielpro/15896s16/slides/896s16-16.pdf

7. Algorithm Design (Tardos and Kleinberg)

Asymptotic Analysis

Debarka Sengupta

Background

What is an Algorithm ?

- An algorithm is a step-by-step procedure or formula for solving a problem, based on conducting a sequence of specified actions.
- An informal definition could be "a set of rules that precisely define a sequence of operations", which would include all computer programs, including programs that do not perform numeric calculations.

History

- Greek mathematicians used algorithms in, for example, the *sieve of Eratosthenes* for finding prime numbers and the *Euclidean algorithm* for finding the greatest common divisor of two numbers.
- The word **algorithm** itself derives from the 9th century mathematician Muḥammad ibn Mūsā al-Khwārizmī, Latinized *Algoritmi.*
- A partial formalization began with attempts to solve the Entscheidungs problem(decision problem) posed by David Hilbert in 1928.
- Later formalizations were framed as attempts to define "effective calculability"[9] or "effective method".

Analysis of Algorithm

- To evaluate rigorously the resources (time and space) needed by an algorithm and represent the result of the evaluation with a formula
- In this course, we focus more on **time** requirement in our analysis
- The time requirement of an algorithm is also called the **time complexity** of the algorithm

Why Algorithm Analysis?

- Predict performance
- Compare algorithms
- Provide guarantees
- Understand theoretical basis

Primary practical reason: avoid performance bugs.

Why not measuring execution time?

- We can measure the actual running time of a program
  - Use **wall clock time** or insert timing code into program

- Simple time calculation gets impacted by
  - CPU speed
  - Different OS - different memory management
  - RAM size
  - Programming language
  - Algorithm implementation

Some huge algorithmic successes

Discrete Fourier transform:

- Break down waveform of N samples into periodic components.
- **Applications:** DVD, JPEG, MRI, astrophysics, ….
- **Brute force:** N^2 steps.
- **FFT algorithm:** $N \log N$ steps, **enables new technology.**

Some huge algorithmic successes

N-body simulation:

- Simulate gravitational interactions among N bodies.
- **Brute force:** N^2 steps.
- **Barnes-Hut algorithm:** $N \log N$ steps, **enables new research**.

Asymptotic analysis - the route

Understanding algorithm

Expressing count of instructions wrt problem size

Understanding asymptotic behavior of the instruction counts

Counting

Counting Operations

- Instead of measuring the actual timing, we count the number of operations.
  - Operations: arithmetic, assignment, comparison, etc.
- Counting an algorithm’s operations is a way to assess its efficiency.
- An algorithm’s execution time is related to the number of operations it requires.

Example: count number of operations required

```
for (int i = 1; i <= n; i++)
{
        perform 100 operations; // A

        for (int j = 1; j <= n; j++)

                {
                        perform 2 operations; // B
                }
}
```

Example: count number of operations required

```
for (int i = 1; i <= n; i++)
{
        perform 100 operations; // A

        for (int j = 1; j <= n; j++)

                {
                        perform 2 operations; // B
                }
}
```

Total Ops = A + B $= \sum_{i=1}^{n} 100 + \sum_{i=1}^{n} (\sum_{j=1}^{n} 2)$

$$= 100n + \sum_{i=1}^{n} 2n \quad = 100n + 2n^2 \quad = 2n^2 + 100n$$

How does it help?

- Knowing the number of operations required by the algorithm, we can state that

  Algorithm X takes $2n^2 + 100n$ operations to solve problem of size n

- If the time t needed for one operation is known, then we can state

  Algorithm X takes $(2n^2 + 100n)t$ time units

How does it help?

- However, time *t* is directly dependent on the factors mentioned earlier

  e.g. different languages, compilers and computers

- Instead of tying the analysis to actual time *t*, we can state

  Algorithm X takes time that is proportional to $2n^2 + 100n$ for solving problem of size n

How does it help?

- Suppose the time complexity of

  Algorithm A is $3n^2 + 2n + \log n + 1/(4n)$

  Algorithm B is $0.39n^3 + n$

- Intuitively, we know Algorithm A will outperform B

  When solving larger problem, i.e. larger n

- The dominating term $3n^2$ and $0.39n^3$ can tell us approximately how the algorithms perform
- The terms n^2 and n^3 are even simpler and preferred
- These terms can be obtained through asymptotic analysis

Expectation

Informally, a probability distribution defines the relative frequency of outcomes of a random variable - the expected value can be thought of as a weighted average of those outcomes (weighted by the relative frequency). Similarly, the expected value can be thought of as the arithmetic mean of a set of numbers generated in exact proportion to their probability of occurring (in the case of a continuous random variable this isn't exactly true since specific values have probability 0).

$$E(x) = \sum xP(X=x)$$

Example: Linear Search Analysis

Best case: O(1)= Target is in first position of the array

Worst Case: O(n)= Target at Last position or not present

Average Case:

First situation - The searched item is present in the array. It can be present at any location of the array with equal probabilities.

Expected number of operations for first situation

$= 1\times 1/n + 2\times 1/n + 3\times 1/n \ldots\ldots + n\times 1/n$

$= 1/n\ (1+2+3+\ldots\ldots+n)$

$= 1/n \times n(n+1)/2$

$= (n+1)/2$

Example: Linear Search Analysis

Second situation - The searched element is not in array. Futile scan of array amounts to *n* comparisons.

Let's reconcile ...

In absence of any prior understanding of the system, it if fair to assume both situations have equal chances to occur.

Expected number of operations

$= 0.5*n + 0.5*(n+1)/2$

$= n/2 + (n+1)/4$

$= (2n+n+1)/4$

$= (3n+1)/4$

$= 3n/4 + 1/4$

Find the complexity for the below function:

```
void function(int n)
{
    int i = 1, s =1;
    while (s <= n)
    {
        i++;
        s += i;
        printf("*");
    }
}
```

Solution

- We can define the terms ‘s’ according to relation $s_i = s_{i-1} + i$. The value of ‘i’ increases by one for each iteration.
- The value contained in ‘s’ at the ith iteration is the sum of the first ‘i’ positive integers.
- If k is total number of iterations taken by the program, then while loop terminates

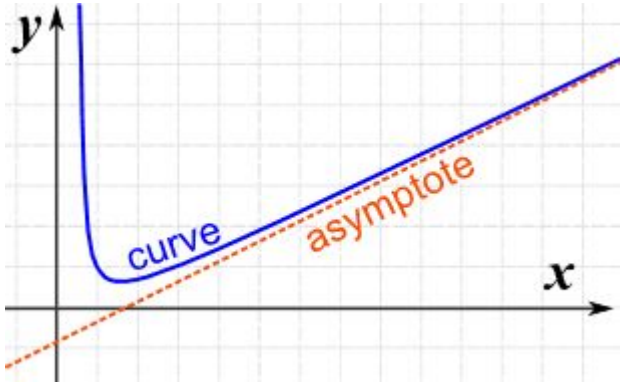
$$\textbf{if: } 1 + 2 + 3 \ldots + k = [k(k+1)/2] > n, \textbf{ So } k = O(\sqrt{n}).$$

- Time Complexity of the above function $O(\sqrt{n})$.

Asymptotic analysis

Asymptote

Put simply, an asymptote is a line (or a curve) that the function keeps getting close to but never actually touches(though we symbolically say it touches it at x = infinity).



Asymptotic Analysis

Asymptotic analysis is an analysis of algorithms that focuses on:

- Analyzing problems of large input size
- Consider only the leading term of the formula
- Ignore the coefficient of the leading term

Leading Term

- Lower order terms contribute lesser to the overall cost as the input grows larger
- Example
  - $f(n) = 2n^2 + 100n$
  - $f(1000) = 2(1000)^2 + 100(1000)$

    $= 2{,}000{,}000 + 100{,}000$
  - $f(100000) = 2(100000)^2 + 100(100000)$

    $= 20{,}000{,}000{,}000 + 10{,}000{,}000$
- Asymptotic behavior of a function is not dependent on the lower order terms.

Examples: Leading Terms

- **$a(n) = \frac{1}{2}n + 4$**
  - Leading term: $\frac{1}{2}$ n
- **$b(n) = 240n + 0.001n^2$**
  - Leading term: $0.001n^2$
- **$c(n) = n \lg(n) + \lg(n) + n \lg(\lg(n))$**
  - Leading term: $n\lg(n)$
  - Note that $\lg(n) = \log_2(n)$

Why Ignore Coefficient of Leading Term?

- Suppose two algorithms have $2n^2$ and $30n^2$ as the leading terms, respectively
- Although actual time will be different due to the different constants, the **growth rates** of the running time are the same
- Compare with another algorithm with leading term of n^3, the difference in growth rate is a much more dominating factor
- Hence, we can drop the coefficient of leading term when studying algorithm complexity

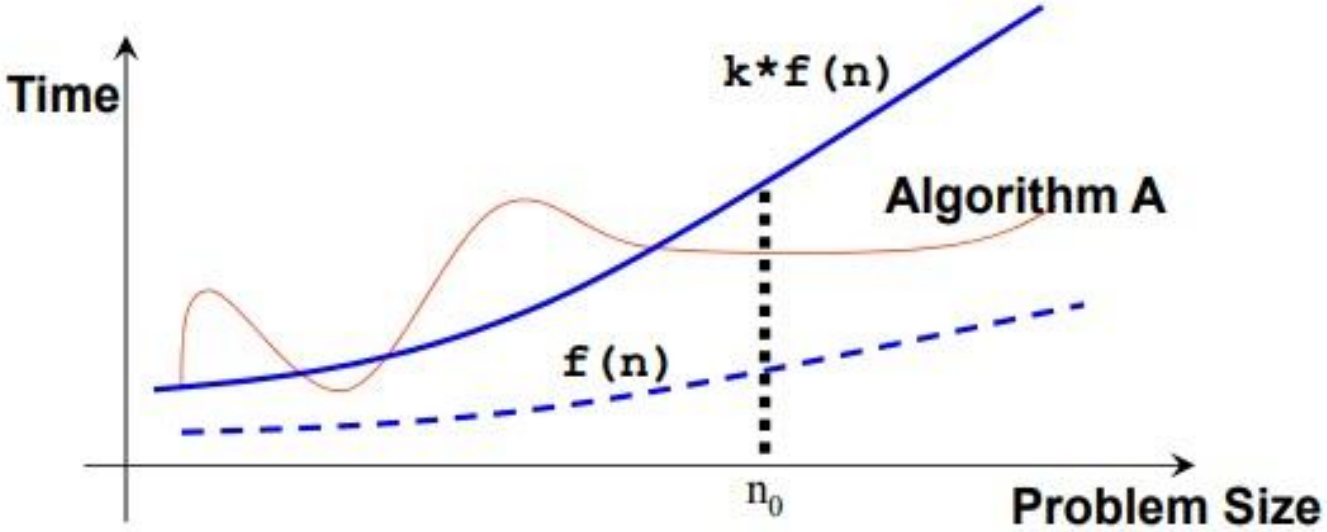
The Big-O Notation

If algorithm *A* requires time proportional to *f(n)*

- Algorithm *A* **is of the order of** *f(n)*
- Denoted as Algorithm A is **O(*f(n)*)**
- *f(n)* is the **growth rate function** for Algorithm *A*

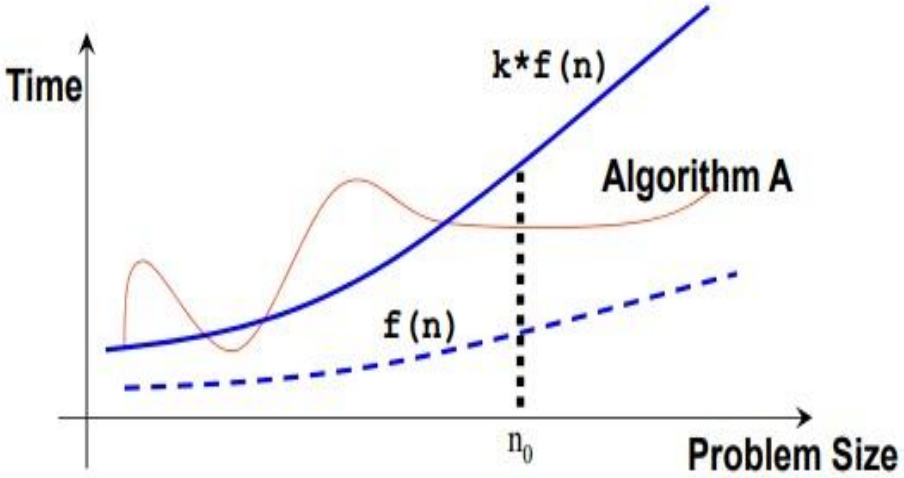
Formal Definition

- Algorithm *A* is of O(*f(n)*) if there exist a constant **k**, and a positive integer $\mathbf{n_0}$ such that Algorithm *A* requires no more than **k*f(n)** time units to solve a problem of size **n >=** $\mathbf{n_0}$



The Big-O Notation

- When problem size is larger than $\mathbf{n_0}$, Algorithm *A* is **bounded from above** by **k * f(n)**
- **Observations :**
  - $\mathbf{n_0}$ and **k** are not unique
  - There are many possible ***f(n)***



Prove

$$F(n) = n^2 + 42n + 7 = O(n^2)$$

Solution sketch

$F(n) = n^2 + 42n + 7 \leq n^2 + 42n^2 + 7n^2 = 50n^2 \ \forall \ n \geq 1$

So $F(n) \leq k.G(n) \ \forall \ n \geq n_0$ where $k = 50$ and $n_0 = 1$

Now we can say $F(n) = O(G(n)) = O(n^2)$

Prove

$$F(n) = 5n.\log_2 n + 8n - 200 = O(n.\log_2 n)$$

Solution hint

$$F(n) = 5n.\log_2 n + 8n - 200 \leq 5n.\log_2 n + 8n$$

$$\leq 5n.\log_2 n + 8n.\log_2 n \ (\forall \ n \geq 2)$$

$$\leq 13n.\log_2 n \ (\forall \ n \geq 2)$$

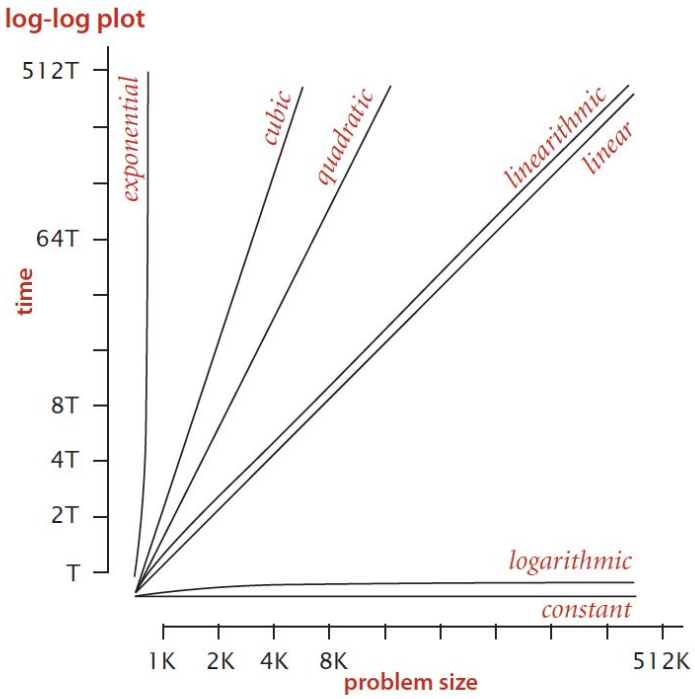
Prove

$$O(\ln n) = O(\log_2 n)$$

Solution hint

$\mathbf{Log_a n = Log_c n / Log_c a = constant \times log_c n}$

Order of growth functions



Typical orders of growth

- **O(1) — constant time**
  - Independent of n
- **O(n) — linear time**
  - Grows as the same rate of n
  - E.g. double input size -> double execution time
- **O(n^2) — quadratic time**
  - Increases rapidly w.r.t. n
  - E.g. double input size -> quadruple execution time
- **O(n^3) — cubic time**
  - Increases even more rapidly w.r.t. n
  - E.g. double input size -> 8 * execution time
- **O(2^n) — exponential time**
  - Increases very very rapidly w.r.t. n

CS 31: Algorithms (Spring 2019): Lecture 3

Date: 28th March, 2019

Topic: Divide and Conquer – Merge Sort, Recurrences, Counting Inversions

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please email errors to deeparnab@dartmouth.edu.

In the next few lectures we will look at the *divide and conquer* paradigm for algorithm design. The problems we will run this technique on will have "naive" or "first-try" algorithms which are polynomial time, but the divide-and-conquer paradigm will lead to significantly faster algorithms.

The methodology is something we already encountered in Lecture 1: when given a problem, we *break/divide* it into one or more smaller subproblems, *recursively* solve the smaller sub-problems, and then combine/conquer the solutions to get the solution to the original problem at hand. The proof of correctness will follow from the way we set it up, and how we handle the base cases. The running time analysis involves solving recurrence inequalities.

Let's start with an algorithm you have seen before.

1 Merge Sort

SORTING AN ARRAY
Input: An array $A[1:n]$ of integers.
Output: A sorted (non-decreasing) permutation $B[1:n]$ of $A[1:n]$.
Size: The number n of entries in A.

Remark: *It is fair to ask why the size of the problem above doesn't include the number of bits required to encode $A[j]$'s and x. The short answer is that it is a modeling choice. If the numbers $A[j]$'s are "small", that is, they are at most some polynomial in n and thus fit in $O(\log n)$ sized registers, then we assume that simple operations such as adding, multiplying, dividing, comparing, reading, writing take $\Theta(1)$ time. The reason being that for numbers that fit in word/registers indeed these operations are fast compared to the other operations of the algorithm.*

Before we go into the divide-and-conquer algorithm for sorting, let us discuss the naive algorithm. Here is one: scan the whole array to find the minimum element, set it to $B[1]$ and remove it from A. Repeat the above process $n-1$ times more. The jth scan takes $(n-j)$ time, and thus the total time taken is $\Theta(n^2)$. Quadratic time is not bad; it is definitely better than going over all the $n!$ permutations and choosing the sorted one. But, we can do better using divide-and-conquer.

This algorithm is *merge-sort*. You have perhaps seen this algorithm before (in CS 10 or CS 1), and the idea nicely captures the Divide-and-Conquer strategy.

First we notice if $n = 1$, then we return the same array; this is the base case.

If $n > 1$, then we first *divide* $A[1 : n]$ into two halves: $A[1 : n/2]$ and $A[n/2 + 1 : n]$. We recursively apply the same algorithm to these halves to obtain sorted versions B_1 and B_2. Note that the final answer that we need is a sorted version of $B_1 \cup B_2$. So in the *combine/conquer* step we do exactly this.

At this point we need to figure out a "win": why is sorting $B_1 \cup B_2$ any easier than sorting $A[1 : n]$ to begin with. The fact we exploit is that these B_1 and B_2 individually are sorted. That is why we can in fact sort $B_1 \cup B_2$ way faster than the $\Theta(n^2)$ naive algorithm for $A[1 : n]$. This is the non-trivial part of the algorithm, and once we get a "win" here over the naive algorithm, we will see that we get a win over all.

Combine Step. Let us then recall the Combine procedure of MERGESORT

COMBINE
Input: Two sorted arrays $P[1 : p]$ and $Q[1 : q]$.
Output: Sorted array $R[1 : r]$ of $P \cup Q$, with $r = p + q$.
Size: $p + q$.

This is an iterative algorithm which keeps three pointers i, j, k all set to 1. At each step we compare $P[i]$ and $Q[j]$ and $R[k]$ is set to whichever is smaller. That particular pointer and k are incremented. The process stops when either i reaches $p + 1$ or j reaches $q + 1$ in which case the rest of the other array is appended to R. The algorithm is correct since (a) it is sorted since every number inserted in R is at least as large as the previous number inserted in R, and (b) all the numbers of $P \cup Q$ appear in R. It takes $\Theta(p + q)$ time as each step takes $\Theta(1)$ time, in each step either i or j increments, and so the algorithm is over in $(p + q)$ steps. A formal pseudocode is given below both of the above combine step and the merge sort.

```
procedure COMBINE(P[1 : p], Q[1 : q]):
    ▷ P and Q are sorted; outputs R which is sorted P ∪ Q.
    i = j = k ← 1.
    while i < p + 1 and j < q + 1 do:
        if (P[i] ≤ Q[j]) then:
            R[k] ← P[i]
            i ← i + 1
        else:
            R[k] ← Q[j]
            j ← j + 1
        k ← k + 1
    if i > p then:
        Append rest Q[j : q] to R
    else:
        Append rest of P[i : p] to R
    return R.
```

Merge Step.

```
1: procedure MERGESORT(A[1 : n]):
2:     ▷ Returns sorted order of A[1 : n]
3:     if n = 1 then:
4:         return A[1 : n]. ▷ Singleton Array
5:     m ← ⌊n/2⌋
6:     B_1 ←MERGESORT(A[1 : m])
7:     B_2 ←MERGESORT(A[m + 1 : n])
8:     return COMBINE(B_1, B_2)
```

Theorem 1. MERGESORT takes $O(n \log n)$ time.

Proof. Let $T(n)$ be the worst case running time of MERGESORT on arrays of size n. For small n, the time taken is some constant which is expressed below.

$$T(1) = \Theta(1) \tag{1}$$

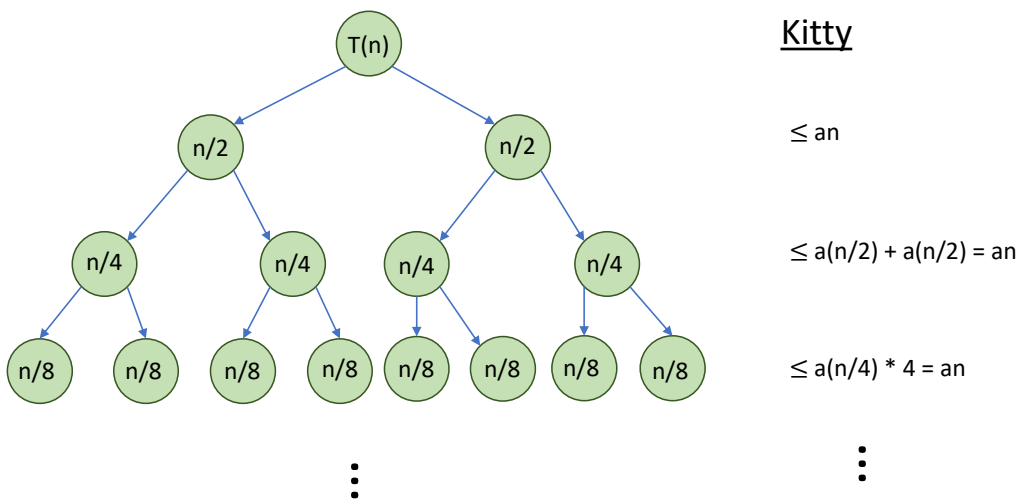
For a given n, let us fix the worst array $A[1 : n]$ which makes MERGESORT run in $T(n)$ time. As we can see the time taken by the algorithm is in Lines 6,7, and 8. These respectively take time at most $T(m)$, $T(n-m)$, and $\Theta(n)$. Noting that $m = \lfloor n/2 \rfloor$, we get

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n), \quad \forall n > 1 \tag{2}$$

To get to the big picture, we get rid of the floors and ceilings. In the supplement, you can see why this is kosher[1]. But for now let's simplify life. Furthermore, we also replace the $\Theta(n)$ term by $\leq a \cdot n$ for some constant a, to get

$$T(n) \leq 2T(n/2) + a \cdot n, \quad \forall n > 1 \tag{3}$$

We will apply the "kitty method"



or "opening up the brackets" method to solve the recurrence inequality given by (1) and (3).

$$\begin{aligned}
T(n) &\leq 2T(n/2) + an \\
&\leq 2\left(2T(n/4) + an/2\right) + an \\
&= 4T(n/4) + 2an \\
&\leq 4\left(2T(n/8) + an/4\right) + 2an \\
&= 8T(n/8) + 3an \\
&\vdots \\
&\leq 2^k T(n/2^k) + kan
\end{aligned}$$

Setting k such that $n/2^k \leq 1$ gives us $T(n) = O(n \log n)$. $\square$

1.1 The Master Theorem

Theorem 2. Consider the following recurrence:

$$T(n) \leq a \cdot T(\lceil n/b \rceil) + \Theta(n^d)$$

[1]if you are a little worried about this, then (a) good, and (b) note that for large x, $\lceil x \rceil, \lfloor x \rfloor$ are really $x \pm$ some "lower order" term, and so since we are talking using the Big-Oh notation, it shouldn't matter.

where a, b, d are non-negative integers. Then, the solution to the above is given by

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

The proof is quite similar to the proof that the recurrence (2) solves to $T(n) \leq O(n \log n)$. Instead of splitting into two, each "ball" splits into a different balls each of size n/b (ignoring floors & ceilings), but it puts $\Theta(n^d)$ in the kitty. If you write the expression as above (it's a little more complicated than the one we say before), then you will get a *geometric series* whose base is exactly a/b^d.

Thus if $a < b^d$ (that is, the base < 1), then the geometric sum is small, and the total cost is bounded by the first deposit in the kitty. If $a = b^d$ (that is, if the base $= 1$) then we make around the same amount of deposits in the kitty, and we do it $\Theta(\log n)$ times. Finally, if $a > b^d$, then the geometric series is bounded by the other end, and the "number of small balls" (which is this bizarrish term $n^{\log_b a}$) is what dominates.

All this is perhaps too high-level to be useful – see the supplement for the proof. ✍

Exercise:

- *Solve the recurrence* $T(n) \leq 2T(n/2) + \Theta(1)$.
- *Solve the recurrence* $T(n) \leq T(n/3) + T(2n/3) + \Theta(n)$.
- *Solve the recurrence* $T(n) \leq \sqrt{n} \cdot T(\sqrt{n}) + \Theta(n)$.

2 Counting Inversions

We now look at a closely related problem. Given an array $A[1:n]$, the pair (i, j) for $1 \leq i < j \leq n$ is called an *inversion* if $A[i] > A[j]$. For example, in the array $[10, 20, 30, 50, 40]$, the pair $(4, 5)$ is an inversion.

<u>COUNTING INVERSION</u>
Input: An array $A[1:n]$
Output: The number of inversions in A.
Size: n, the size of the array.

There is a naive $O(n^2)$ time algorithm: go over all pairs and check if they form an inversion or not. We now apply the divide-and-conquer paradigm to do better.

If $n = 1$, then the number of inversions is 0. Otherwise, suppose we divide the array into two: $A[1:n/2]$ and $A[n/2+1:n]$. Recursively, suppose we have computed the number of inversions in $A[1:n/2]$ and $A[n/2+1:n]$. Let these be I_1 and I_2, respectively. Note that any inversion (i, j) in $A[1:n]$ satisfies either
(a) $i < j \leq n/2$, which implies (i, j) is an inversion in $A[1:n/2]$, or

(b) $n/2+1 \le i < j$, which implies (i,j) is an inversion in $A[n/2+1:n]$, or
(c) $i \le n/2 < j$, and these are the extra inversions over $I_1 + I_2$ that we need to count.

Let's call any (i,j) of type (c) above a *cross* inversion, and let C denote this number. Then by what we said above, we need to return $I_1 + I_2 + C$. Is it any easier to calculate C?

After you think about it for a while, there may not seem to be any easy way to calculate C faster than $O(n^2)$. There are two crucial observations that help here.

- The number of cross-inversions between $A[1:n/2]$ and $A[n/2+1:n]$ is the same as between $\mathsf{sort}(A[1:n/2])$ and $\mathsf{sort}(A[n/2+1:n])$.
- If $A[1:n/2]$ and $A[n/2+1:n]$ were sorted, then the cross-inversions can be calculated in $O(n)$ time. This may not be immediate, but if you understand the COMBINE subroutine above, then it should ring a bell. We elaborate it on it later.

Cross-Inversions between Sorted Arrays. Given two sorted arrays $P[1:p]$ and $Q[1:q]$, we can count the number of cross-inversion pairs (i,j) such that $P[i] > Q[j]$ in $O(n)$ time as follows. As in COMBINE we start off with two pointers i, j initialized to 1. We also store a counter num initialized to 0. We check if $P[i] > Q[j]$ or not. If it isn't, then we increment $i = i+1$. Otherwise, we increment $\mathsf{num} = \mathsf{num} + (p-i+1)$ and $j = j+1$. The claim below explains why we increment the num as we do. We stop when either $i = p+1$ or $j = q+1$.

Claim 1. At any stage, suppose the algorithm encounters $P[i] > Q[j]$. Then $\{(i',j) : i' \ge i\}$ are the only cross-inversions which involve j.

Proof. Since P is sorted, $P[i'] > Q[j]$ for all $i' \ge i$ and so all such (i',j) are inversions. Now consider any $i'' < i$. Since in the algorithm the pointer is at $i > i''$, at some previous stage the algorithm compared $P[i'']$ and $Q[j'']$ with $j'' \le j$, and found $P[i''] \le Q[j'']$. But since Q is sorted, this would imply $P[i''] \le Q[j]$. This implies for all $i'' < i$, (i'',j) is not an inversion. □

```
procedure COUNTCROSSINV(P[1:p], Q[1:q]):
    ▷ P and Q are sorted; outputs the number of (i,j) with P[i] > Q[j].
    i ← 1; j ← 1; num ← 0.
    while i < p+1 and j < q+1 do:
        if (P[i] > Q[j]) then:
            num ← num + (p − i + 1)
            j ← j + 1
        else:
            i ← i + 1
```

Now we are armed to describe the divide-and-conquer algorithm for counting inversions.

```
1:  procedure CountInv1(A[1 : n]):
2:      ▷ Counts the number of inversions in A[1 : n]
3:      if n = 1 then:
4:          return 0. ▷ Singleton Array
5:      m ← ⌊n/2⌋
6:      I_1 ←CountInv1(A[1 : m])
7:      I_2 ←CountInv1(A[m + 1 : n])
8:      B_1 ←MergeSort(A[1 : m])
9:      B_2 ←MergeSort(A[m + 1 : n])
10:     C ←CountCrossInv(B_1, B_2)
11:     return I_1 + I_2 + C.
```

Let's analyze the time complexity. As always, let $T(n)$ be the worst case running time of CountInv1 on an array of length n. Let $A[1 : n]$ be the array attaining this time, and let's see the run of the algorithm on this array. The time taken by Lines 6 and 7 are $T(\lfloor n/2 \rfloor)$ and $T(\lceil n/2 \rceil)$ respecively. The time taken by Line 10 takes $\Theta(n)$ time by what we described above. Furthermore, the Lines 8 and 9 takes $\Theta(n \log n)$ time. Together, we get the following recurrence

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n \log n)$$

The above 'almost' looks like (2), and indeed the recurrence solves to $T(n) = \Theta(n \log^2 n)$.

But there is something wasteful about the above algorithm. In particular, if you run it on a small example by hand you will see that you are sorting a lot. And often the same sub-arrays. Is there some way we can exploit this and get a faster algorithm?

You should solve this and check yourself.

Indeed we can. This introduces a new idea in the divide-and-conquer paradigm: *we can get more by asking for more*. This "asking for more" technique is something you may have seen while proving statements by induction where you can prove something you want by actually asking to prove something stronger by induction. In this problem, we ask our algorithm to do more: given an array $A[1 : n]$ it has to count the inversions and also has to sort the array too. Now note that in this case Lines 8 and 9 are not needed any more; this is returned by the new stronger algorithm. We however need to also return the sorted array : but this precisely what Combine does. So the final algorithm for counting inversions is below.

```
procedure SORT-AND-COUNT(A[1 : n]):
    ▷ Returns (B, I) where B = sort(A) and I is the number of inversions in A[1 : n]
    if n = 1 then:
        return (A, 0). ▷ Singleton Array
    m ← ⌊n/2⌋
    (B_1, I_1) ← SORT-AND-COUNT(A[1 : m])
    (B_2, I_2) ← SORT-AND-COUNT(A[m + 1 : n])
    C ← COUNTCROSSINV(B_1, B_2)
    B ← COMBINE(B_1, B_2)
    return (B, I_1 + I_2 + C)
```

Now we see that the recurrence for the running time of SORT-AND-COUNT is precisely

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$$

Theorem 3. SORT-AND-COUNT returns the number of inversions of an array $A[1 : n]$ in $\Theta(n \log n)$ time.

Divide and Conquer (Part 1)

Debarka Sengupta

Divide and Conquer

- Reduce problem to one or more sub-problems of the same type
- Typically, each sub-problem is at most a constant fraction of the size of the original problem
- Subproblems typically disjoint
- Often gives significant, usually polynomial, speedup
- Examples: Quick Sort, Mergesort, Binary Search, Strassen's Algorithm, Quicksort. FFT, etc.

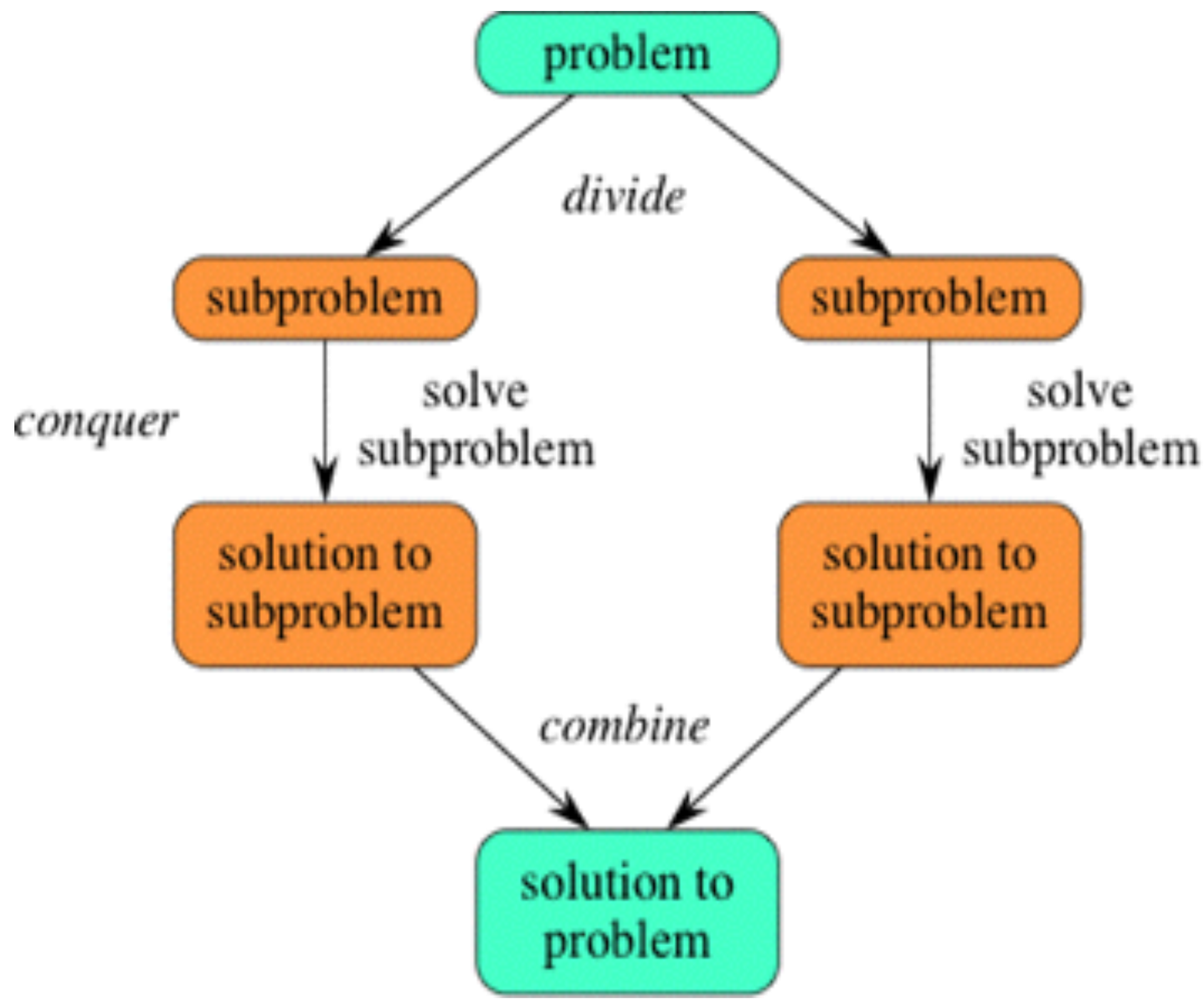
Steps involved

- Divide-and-conquer algorithm has three parts:
- Divide: the problem into a number of subproblems that are smaller instances of the same problem.
- Conquer: the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
- Combine: the solutions to the subproblems into the solution for the original problem.Consequence:

Benefit

- Brute force / naïve solution: N^2 (typically)
- Divide-and-conquer: N log N

Algorithm sketch - two subproblems



Algorithm sketch - more than two subproblems

Finding word in dictionary

Suppose you want to find **“janissary”** in a dictionary:

- open the book near the middle
- the heading on the top left page is **“kiwi”**,
- so move back a small number of pages
- here you find **“hypotenuse”**, so move forward
- find **“ichthyology”**, move forward again

The number of pages you move gets smaller (or at least adjusts in response to the words you find)

Common D&C algorithms

- **Mathematics**
  - Polynomial & Matrix Multiplication
  - Exponentiation
  - Large Integer Manipulation
  - FFT
- **Geometry**
  - Convex Hull
  - Closest Pair
- **Searching**
  - Binary Search
- **Sorting**
  - Merge Sort
  - Quick Sort

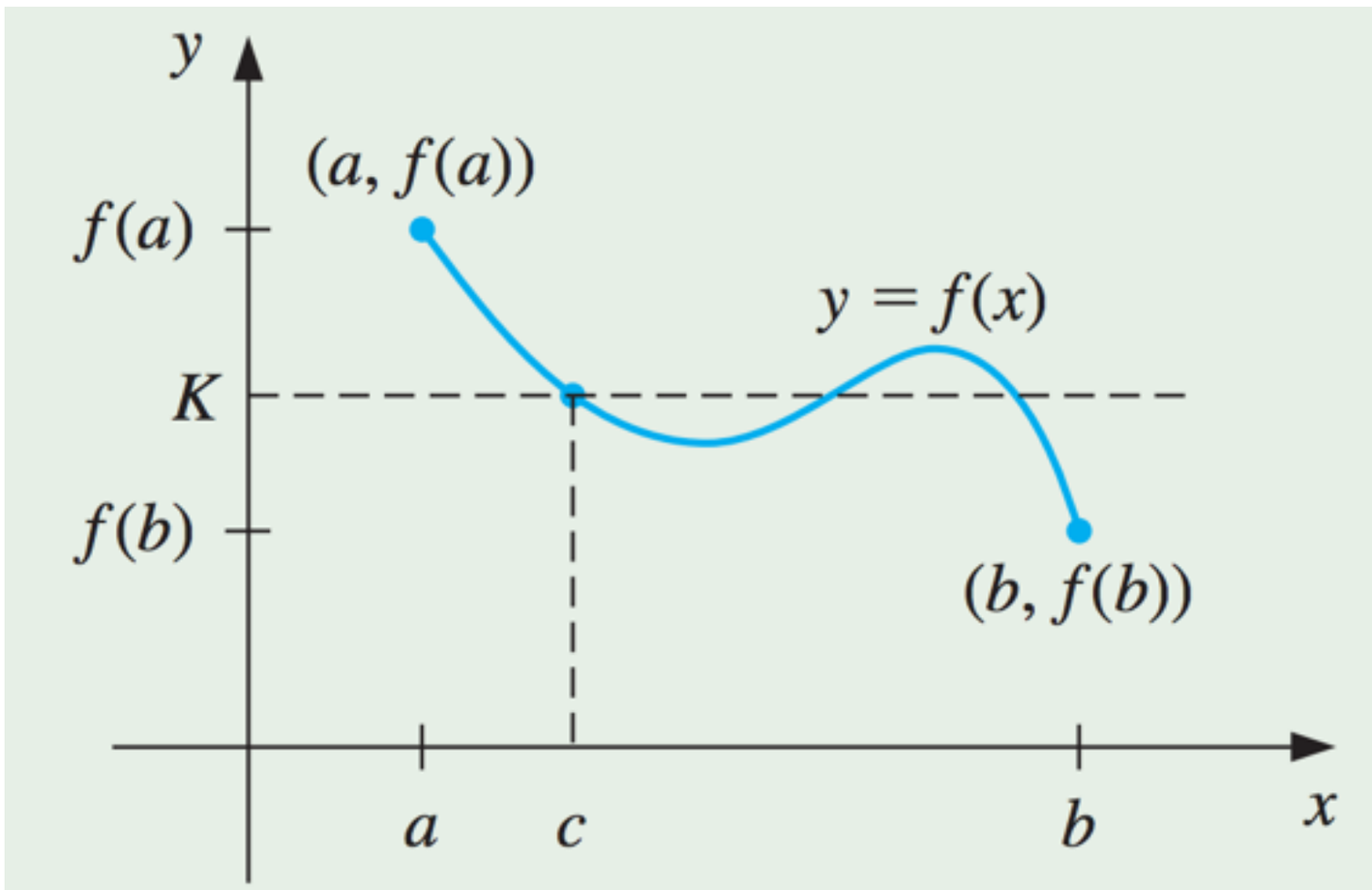
Root finding - bisection

Root finding problem

- Finding Zero of function f(x)
- This process involves finding a root, or solution, of an equation of the form f(x) = 0 for a given function f

Intermediate value theorem (IVT)

It simply states that for any value L between f(a) & f(b), there's a value c in [a,b] for which f(c)=L.



Example

- Show that $f(x) = x^3 + 4x^2 - 10 = 0$ has a root in [1, 2] and use the Bisection method to determine an approximation to the root that is accurate to at least within 10^{-4}

Solution sketch

- Because $f(1) = -5$ and $f(2) = 14$ the IVT ensures that this continuous function has a root in $[1, 2]$.

Solution sketch continued …

- For the first iteration of the Bisection method we use the fact that at the midpoint of [1, 2] we have $f(1.5) = 2.375 > 0$.

- This indicates that we should select the interval [1, 1.5] for our second iteration.

- Then we find that $f(1.25) = -1.796875$ so our new interval becomes [1.25, 1.5], whose midpoint is 1.375.

- Continuing in this manner gives the values shown in the following table

Iterations

Iter	a_n	b_n	p_n	$f(a_n)$	$f(p_n)$	RelErr
1	1.000000	2.000000	1.500000	-5.000	2.375	0.33333
2	1.000000	1.500000	1.250000	-5.000	-1.797	0.20000
3	1.250000	1.500000	1.375000	-1.797	0.162	0.09091
4	1.250000	1.375000	1.312500	-1.797	-0.848	0.04762
5	1.312500	1.375000	1.343750	-0.848	-0.351	0.02326
6	1.343750	1.375000	1.359375	-0.351	-0.096	0.01149
7	1.359375	1.375000	1.367188	-0.096	0.032	0.00571
8	1.359375	1.367188	1.363281	-0.096	-0.032	0.00287
9	1.363281	1.367188	1.365234	-0.032	0.000	0.00143
10	1.363281	1.365234	1.364258	-0.032	-0.016	0.00072
11	1.364258	1.365234	1.364746	-0.016	-0.008	0.00036
12	1.364746	1.365234	1.364990	-0.008	-0.004	0.00018
13	1.364990	1.365234	1.365112	-0.004	-0.002	0.00009

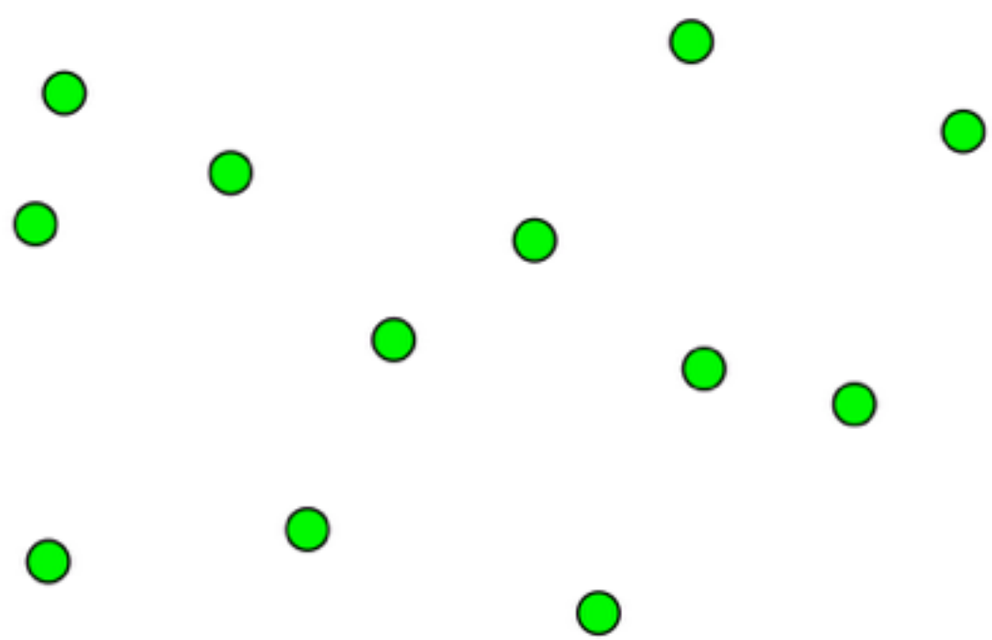
Precision

- The number of iterations depends on the precision being looked for
- if $c_1 = (a+b)/2$ is the midpoint of the initial interval, and c_n is the midpoint of the interval in the nth step, then the difference between c_n and a solution c is bounded by $|c_n - c| <= |b - a|/2^n$
- One should be able to perform the back calculation

Closest Pair

Closest pair - problem statement

- **Given a set of points $\{p_1, \ldots, p_n\}$ find the pair of points $\{p_i, p_j\}$ that are closest**

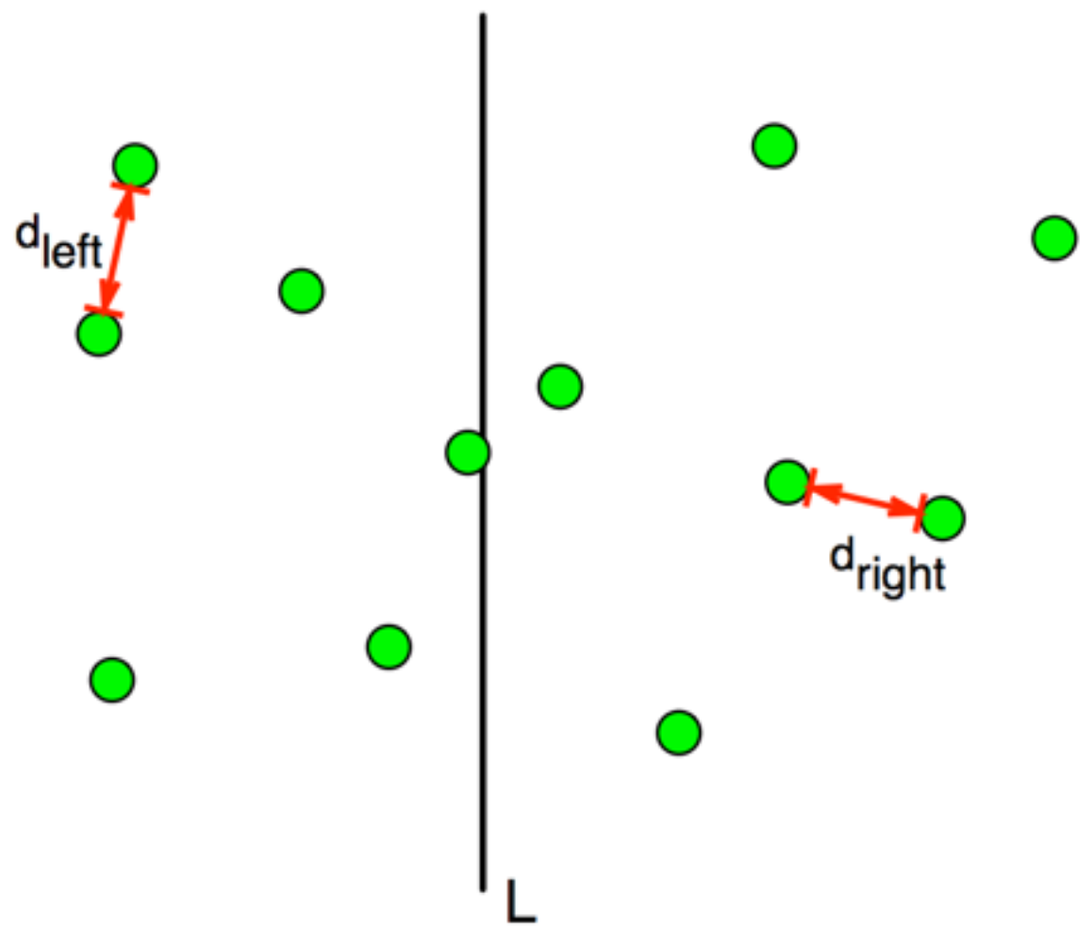


Brute force vs. divide and conquer

- Brute force gives an $O(n^2)$ algorithm to just check ever pair of points.
- Can we do it faster? Seems like difficult without checking every pair.
- In fact, we can find the closest pair in $O(n \log n)$ time.

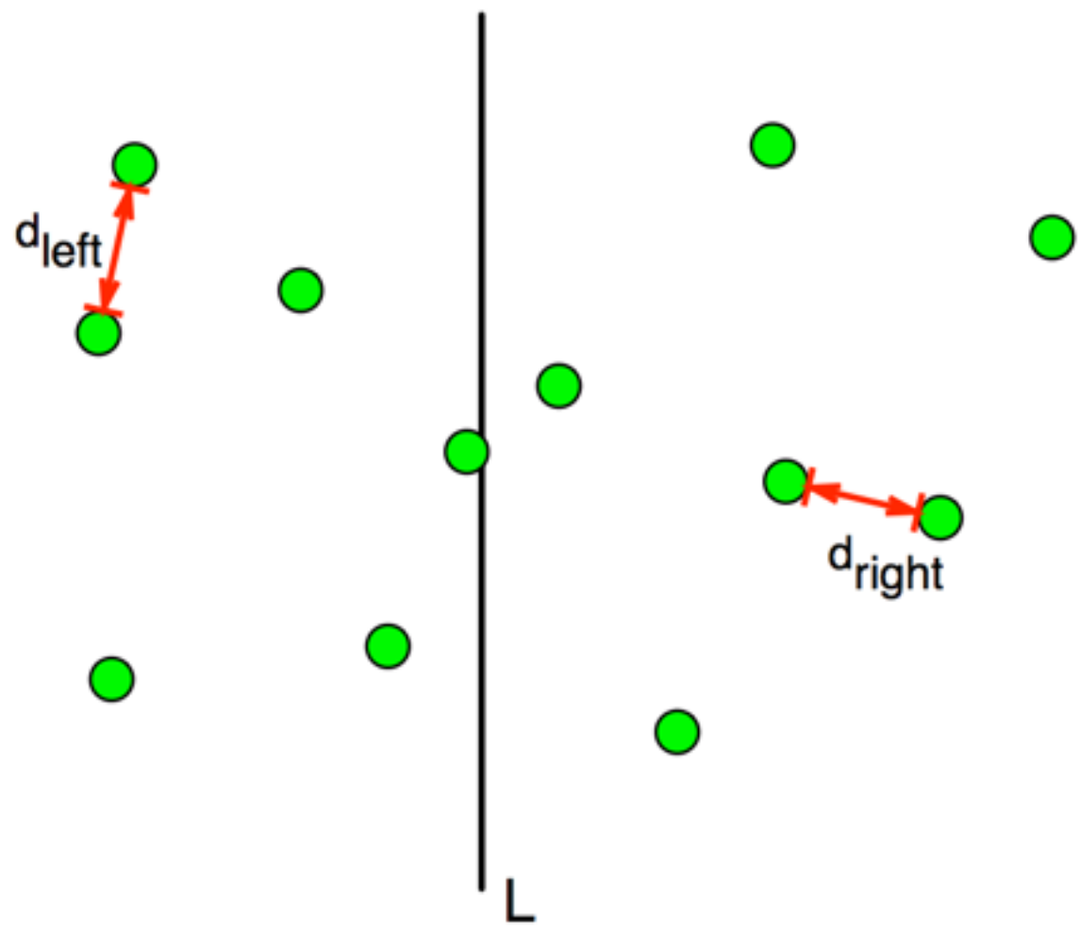
Divide

- **Split the points with line L so that half the points are on each side.**
- **Recursively find the pair of points closest in each half.**



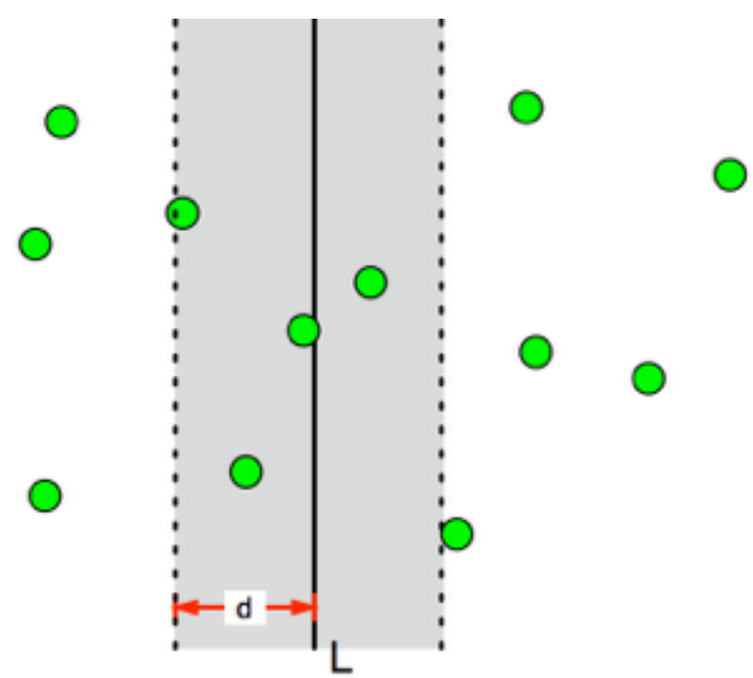
Merge

- **Let $d = \min\{d_{left}, d_{right}\}$**
- **d would be the answer, except maybe L split a close pair!**



Region near L

- **If the closest pair exists across the L, it should contained within d margins on both sides**



A life saver observation

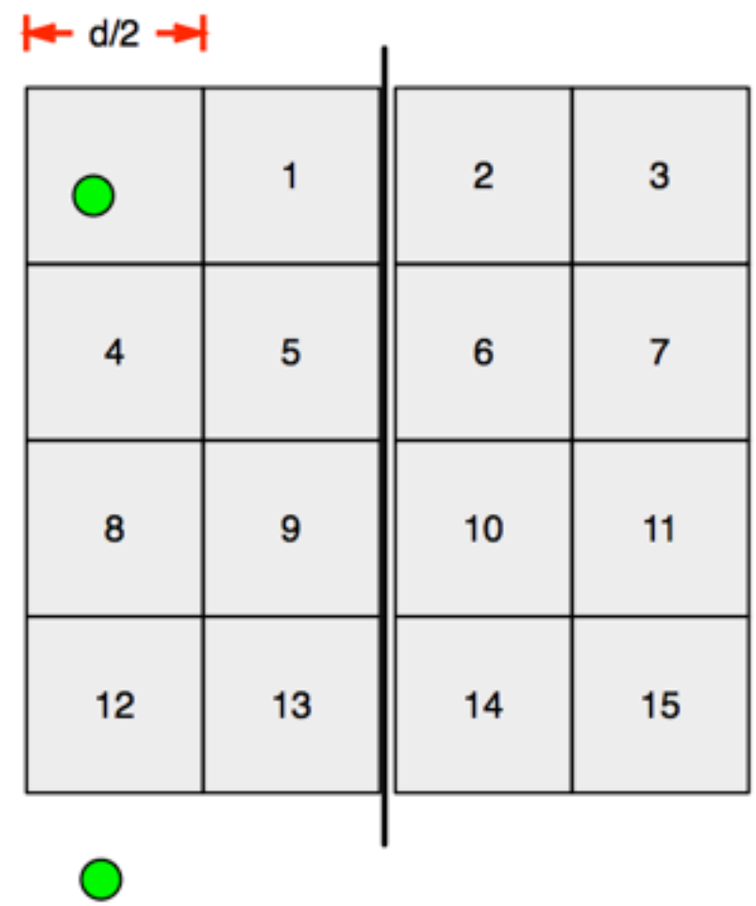
- Let S_y be an array of the points in that region, sorted by decreasing y-coordinate value.
- S_y might contain all the points, so we can't just check every pair inside it.

Theorem

Suppose $S_y = p_1, \ldots, p_m$. *If* $dist(p_i, p_j) < d$ *then* $j - i \leq 15$.

Constant number of checks for each point is enough

- We can split the entire margin area into d/2 sided square boxes
- Each box can have max 1 point
- Suppose 2 points are separated by > 15 indices.
  - Then, at least 3 full rows separate them (the packing shown is the smallest possible).
  - But the height of 3 rows is > 3d/2, which is > d.
  - So the two points are further than d apart.



Conclusion

- Starting from the bottom most point, for each point, it is enough to check next 15 data points up in the sequence of increasing value in S_y
- Constant (15) time for each of the n points

The algorithm

```
ClosestPair(Px, Py):
   if |Px| == 2: return dist(Px[1],Px[2])     // base

   d1 = ClosestPair(FirstHalf(Px,Py))      // divide
   d2 = ClosestPair(SecondHalf(Px,Py))
   d = min(d1,d2)

   Sy = points in Py within d of L      // merge
   For i = 1,...,|Sy|:
       For j = 1,...,15:
           d = min( dist(Sy[i], Sy[j]), d )
   Return d
```

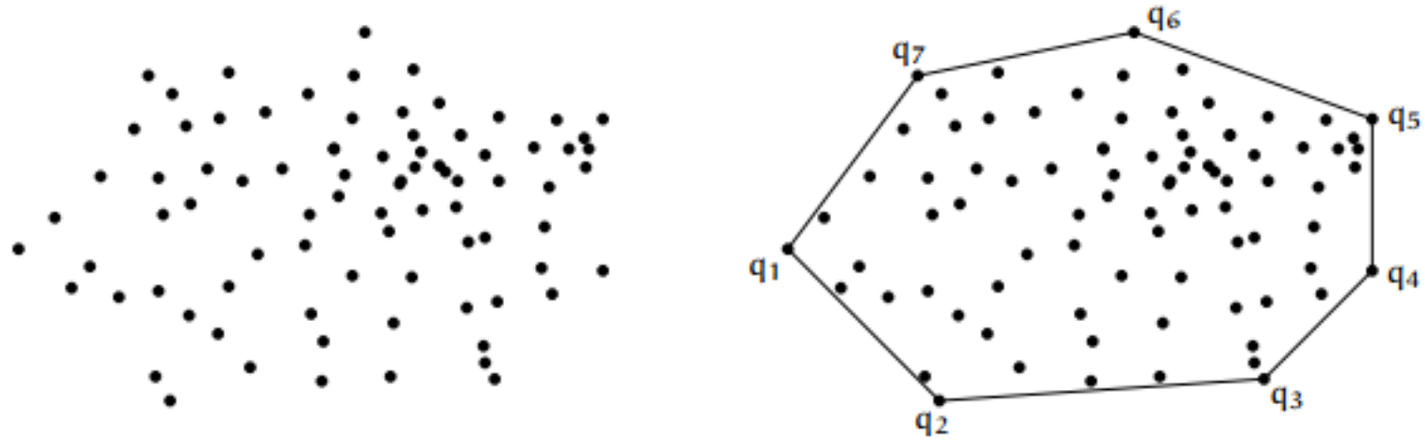
Running time

- Divide set of points in half each time: O(log n) depth recursion
- Merge takes O(n) time.
- Recurrence: $T(n) = 2T(n/2) + cn$
- Analysis is same as MergeSort ⇒ O(n log n) time

Convex Hull

Convex Hull

- Studied in the field of Computational Geometry.
- A convex hull, of a set of N points is the smallest perimeter fence enclosing all these points.
- Convex Hull Output - Sequence of vertices in counterclockwise/ clockwise order.



(a) Input. (b) Output.

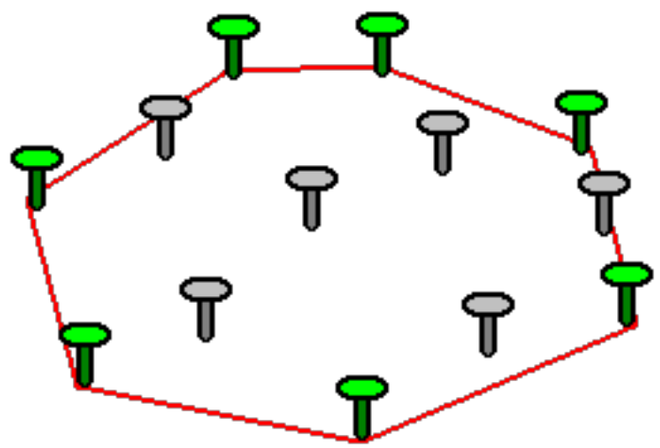
Motion planning

- Robot Motion Planning - Find the shortest path in the plane from a starting point s to an ending point t, that avoids a polygonal obstacle.

- Here, we can see that the shortest path is either the straight line from s to t, or one of the two polygon chains of the convex hull.

Mechanical algorithm

Hammer nails perpendicular to the plane and stretch elastic rubber band around these points.



Brute force

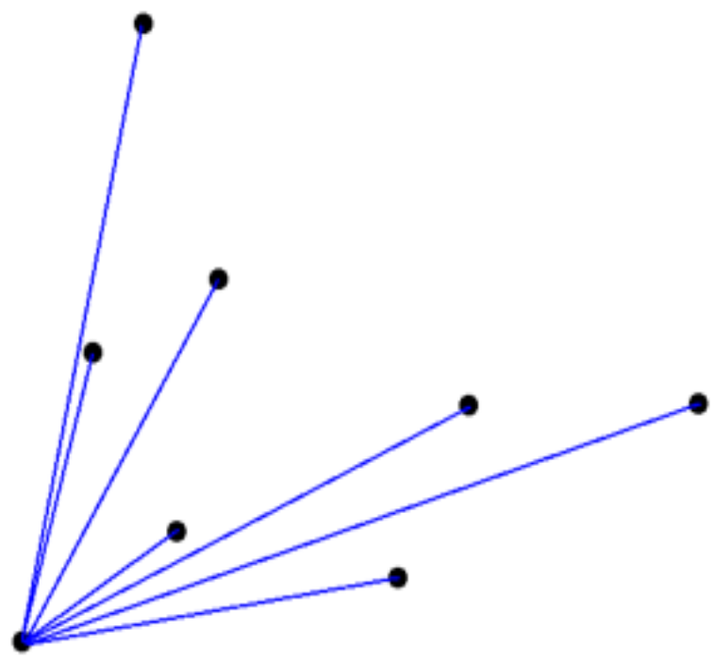
- Create a list of all possible line segments
  - N choose 2 line segments for N points
- For each line segment check if both sides have points
  - If that’s not the case, include the segment in the solution set
- Quadratic complexity algorithm

Graham's scan

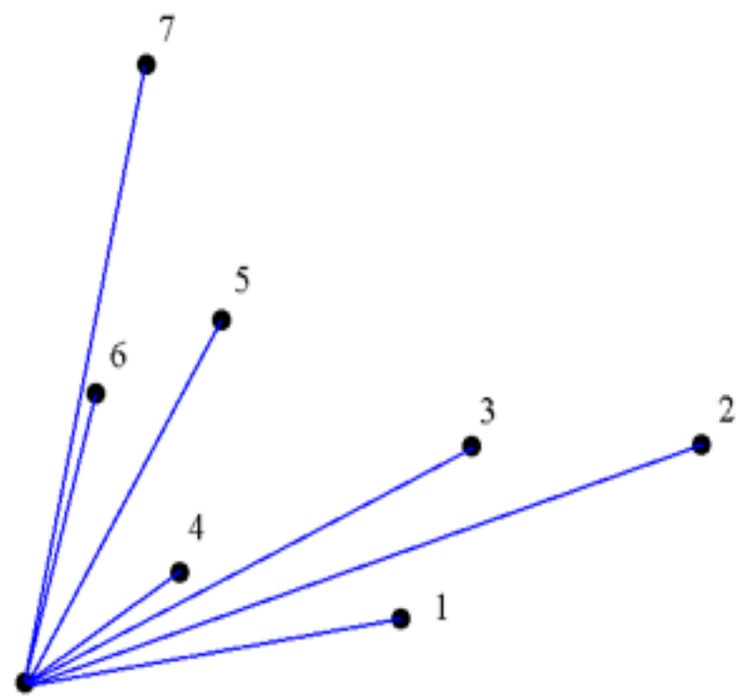
- Start at point guaranteed to be on the hull. (the point with the minimum y value)
- Sort remaining points by polar angles of vertices relative to the first point.
- Go through sorted points, keeping vertices of points that have left turns and dropping points that have right turns.

Graham’s scan example

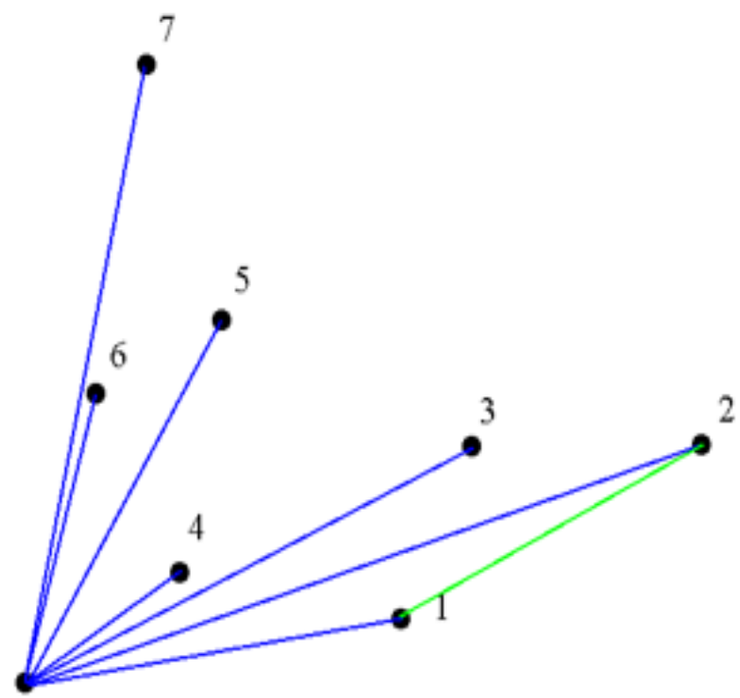
Graham’s scan example



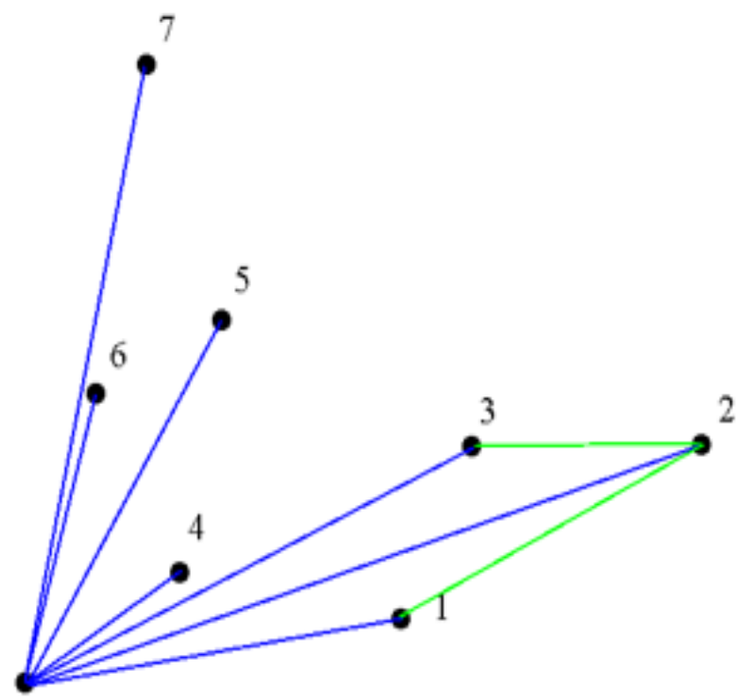
Graham’s scan example



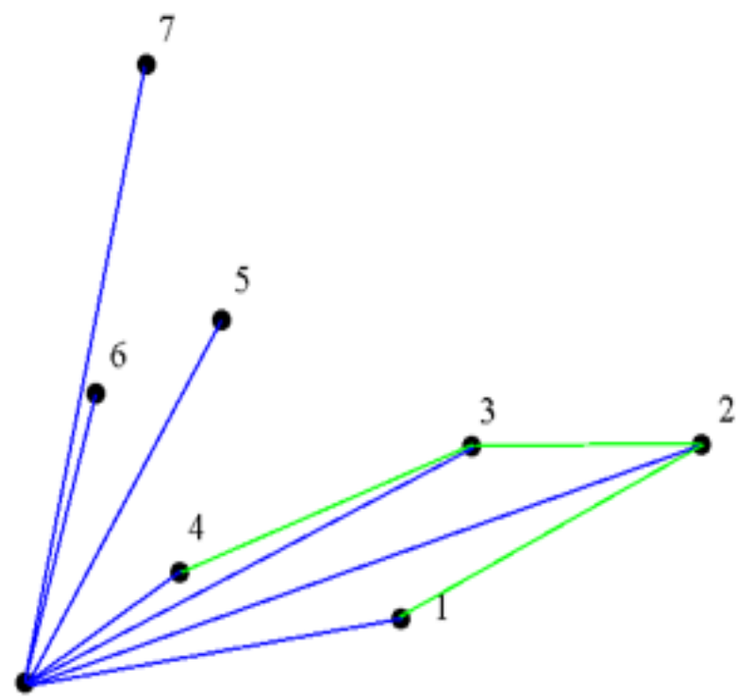
Graham’s scan example



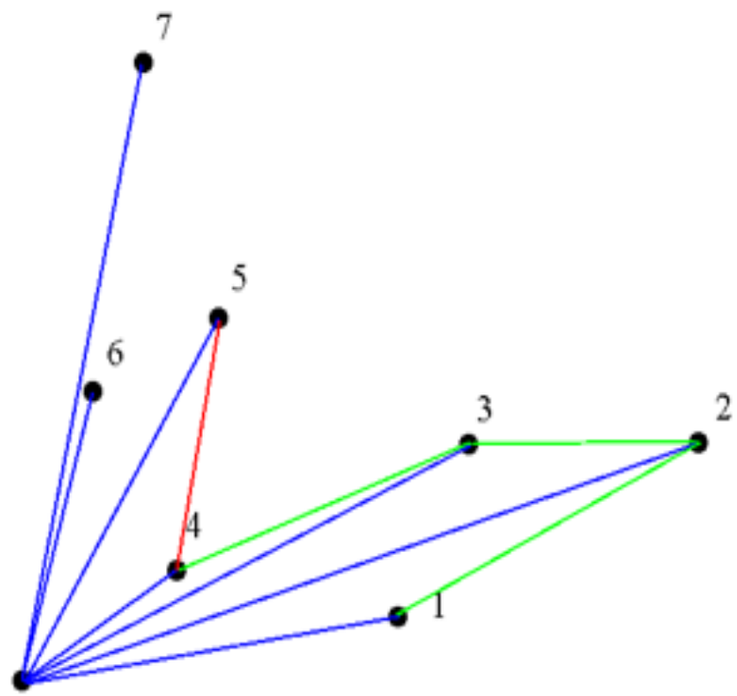
Graham’s scan example



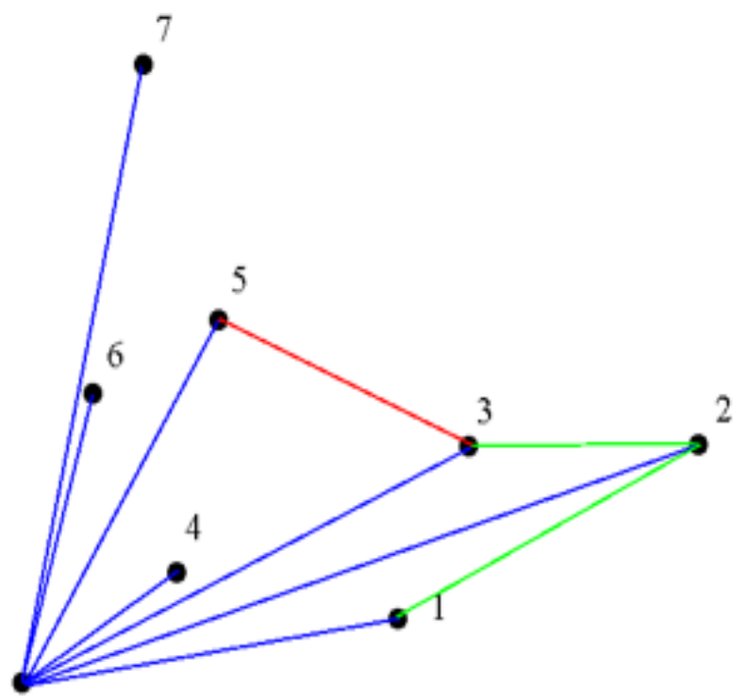
Graham's scan example



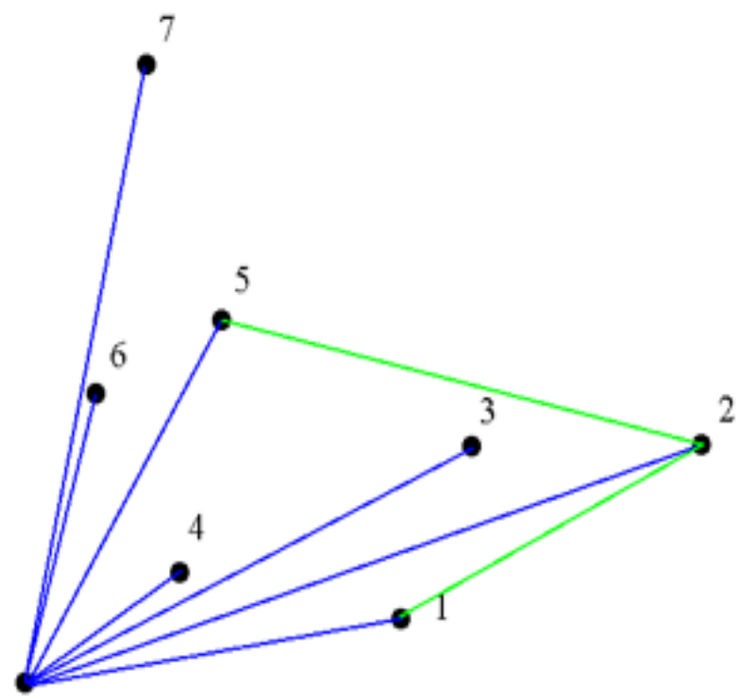
Graham's scan example



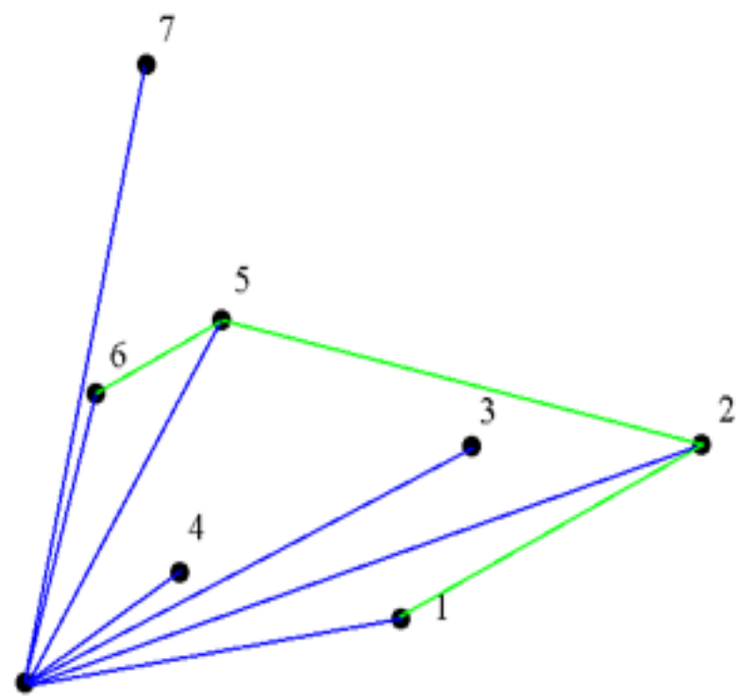
Graham’s scan example



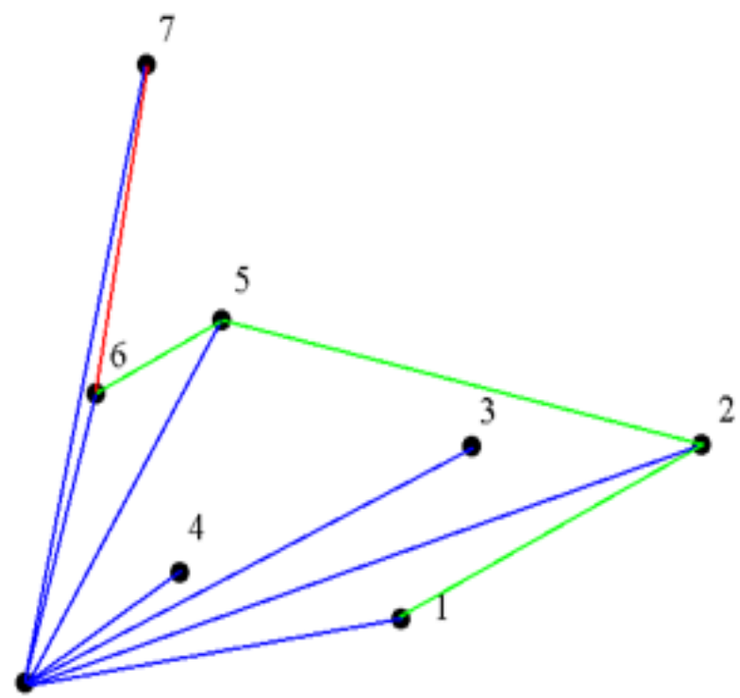
Graham's scan example



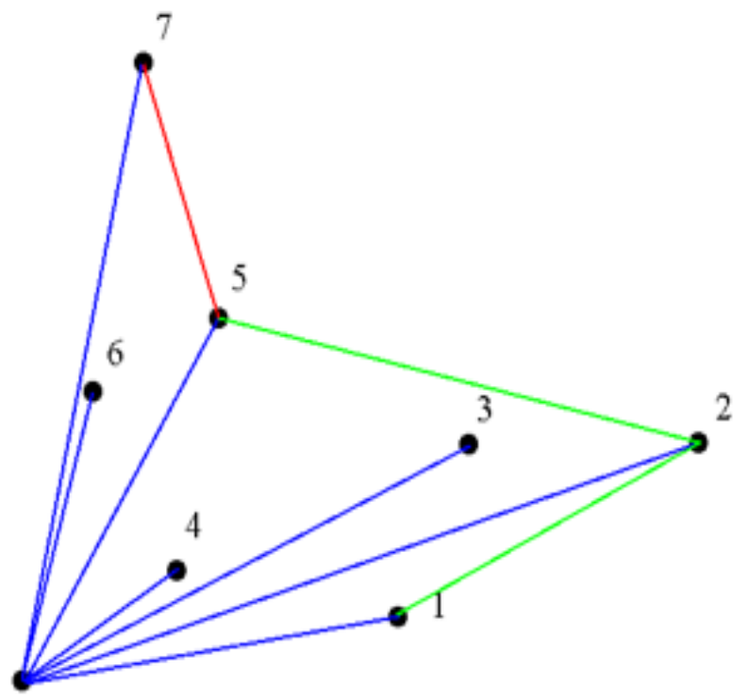
Graham's scan example



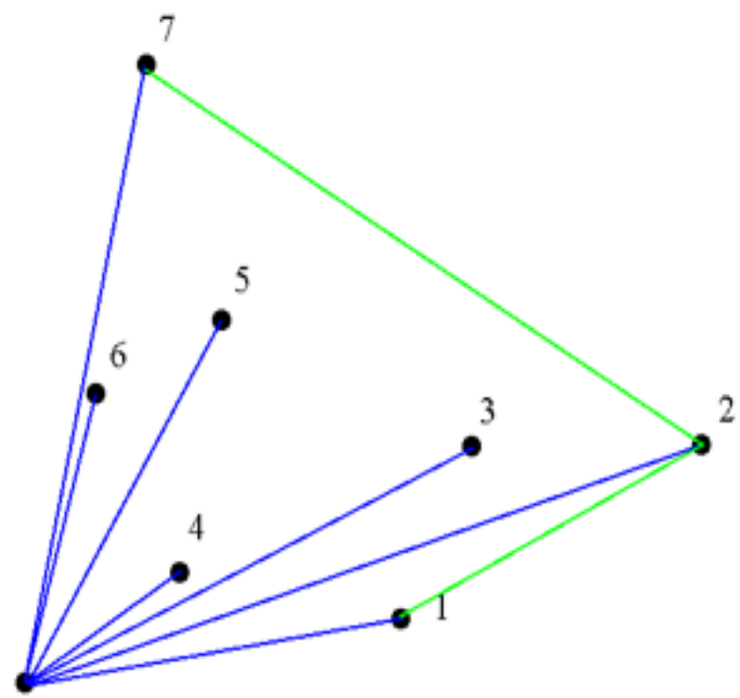
Graham’s scan example



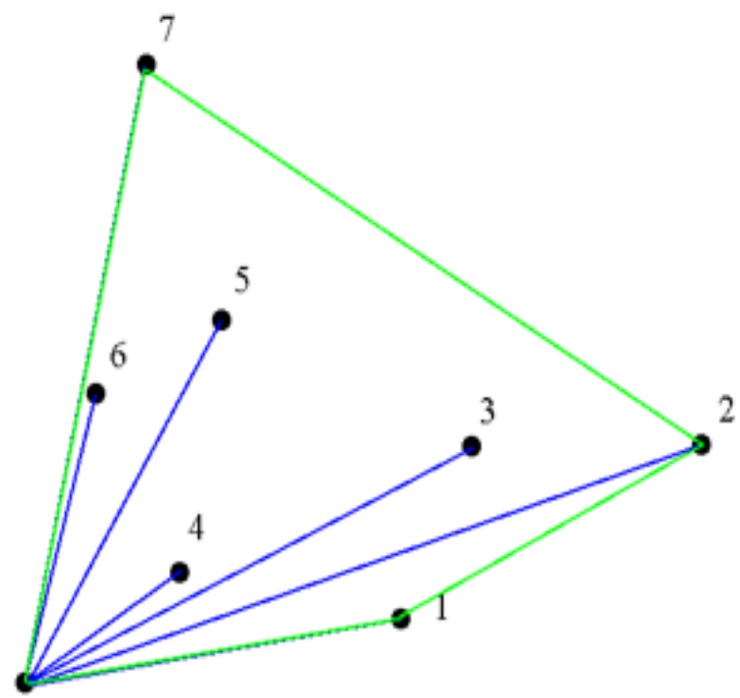
Graham’s scan example



Graham’s scan example



Graham's scan example

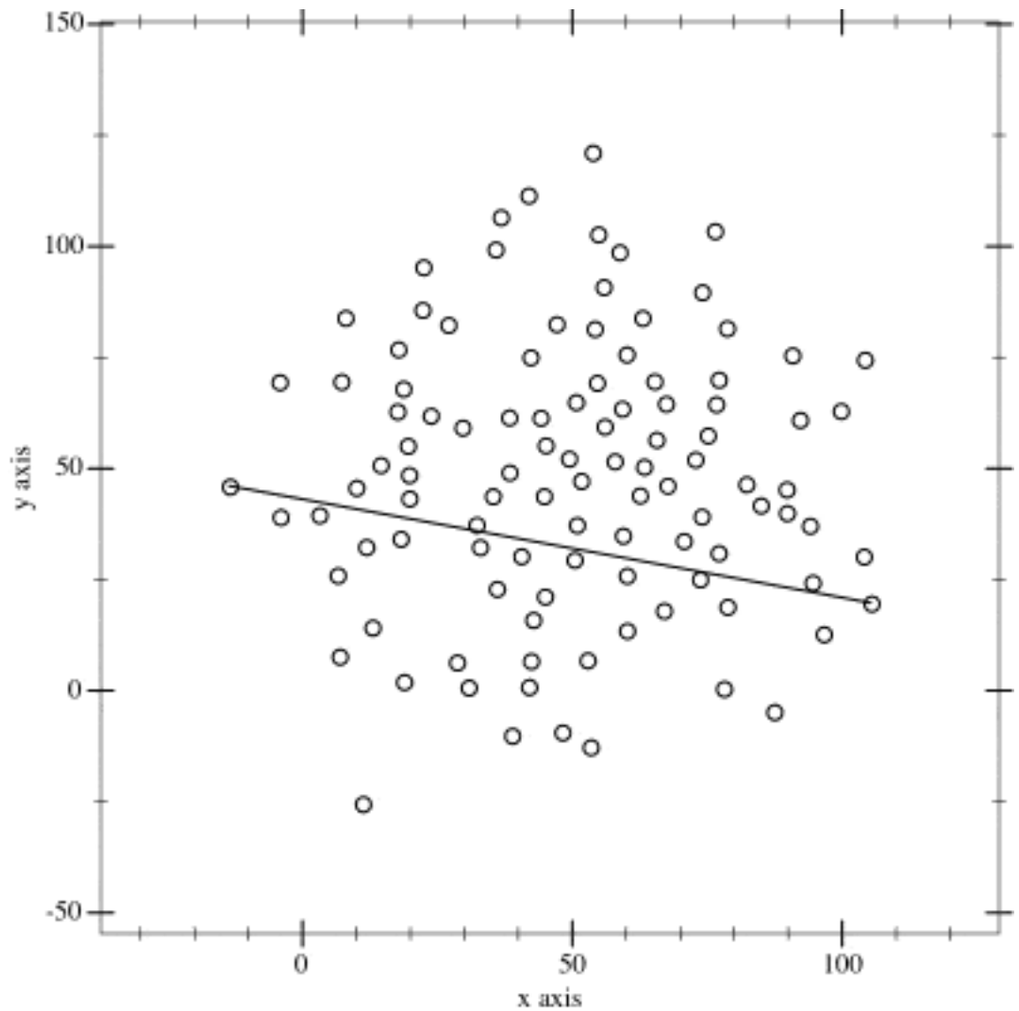


Runtime of Graham's can

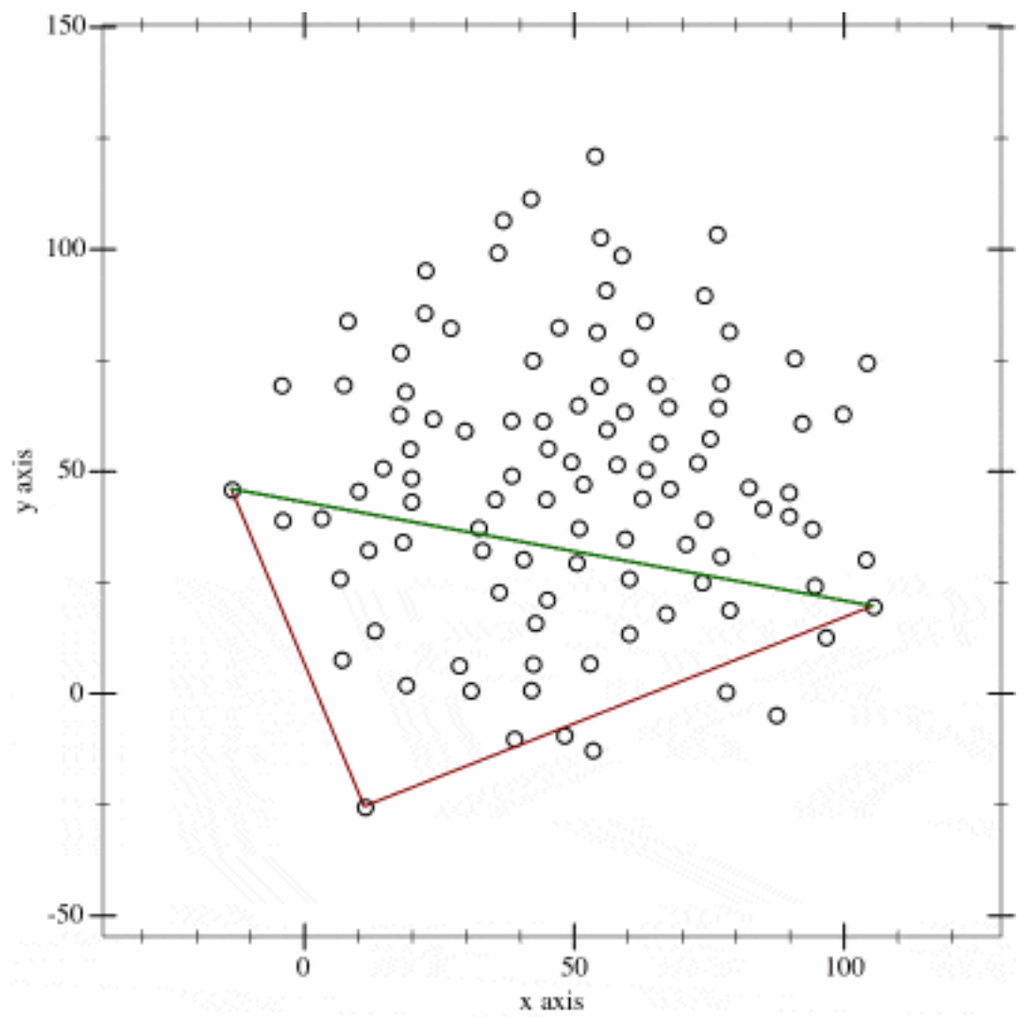
- Graham's scan is O(n log n) due to initial sort of angles.

Quick Hull Algorithm

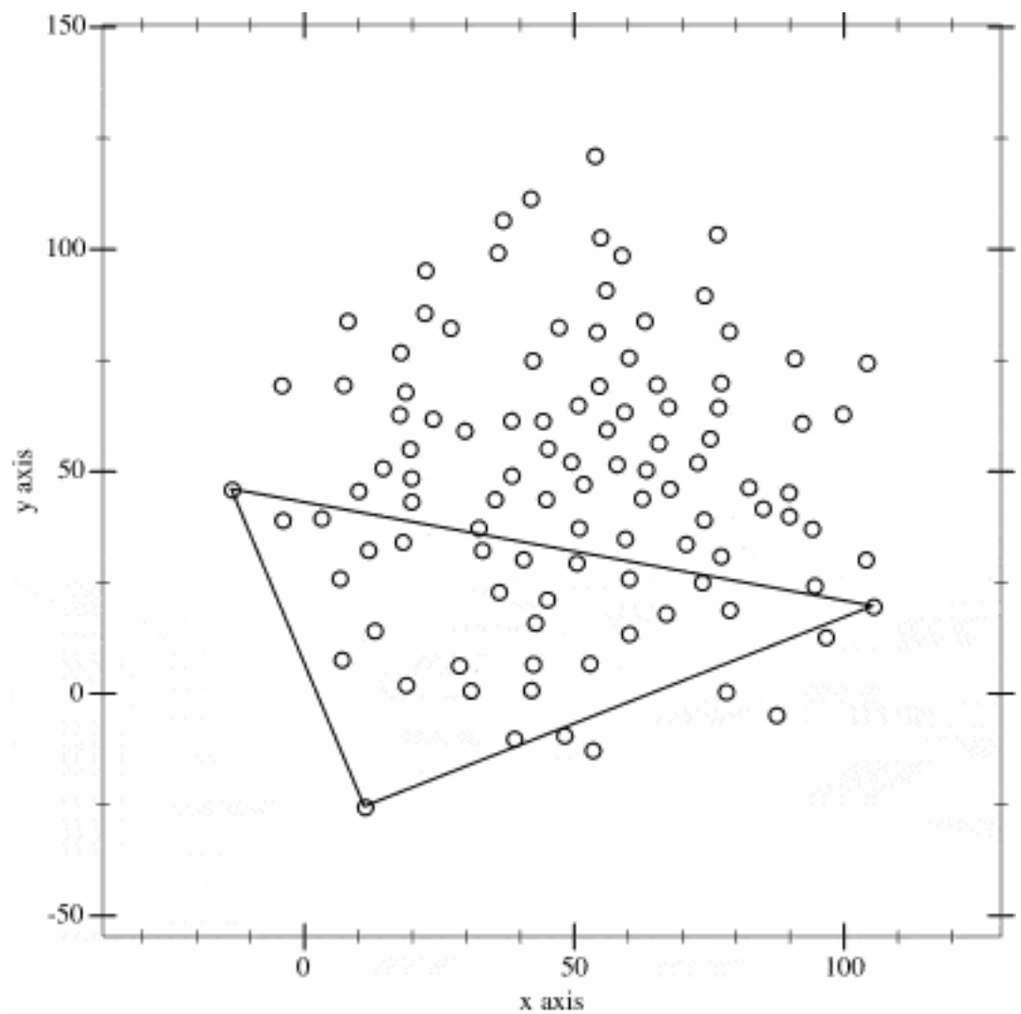
Quick hull example



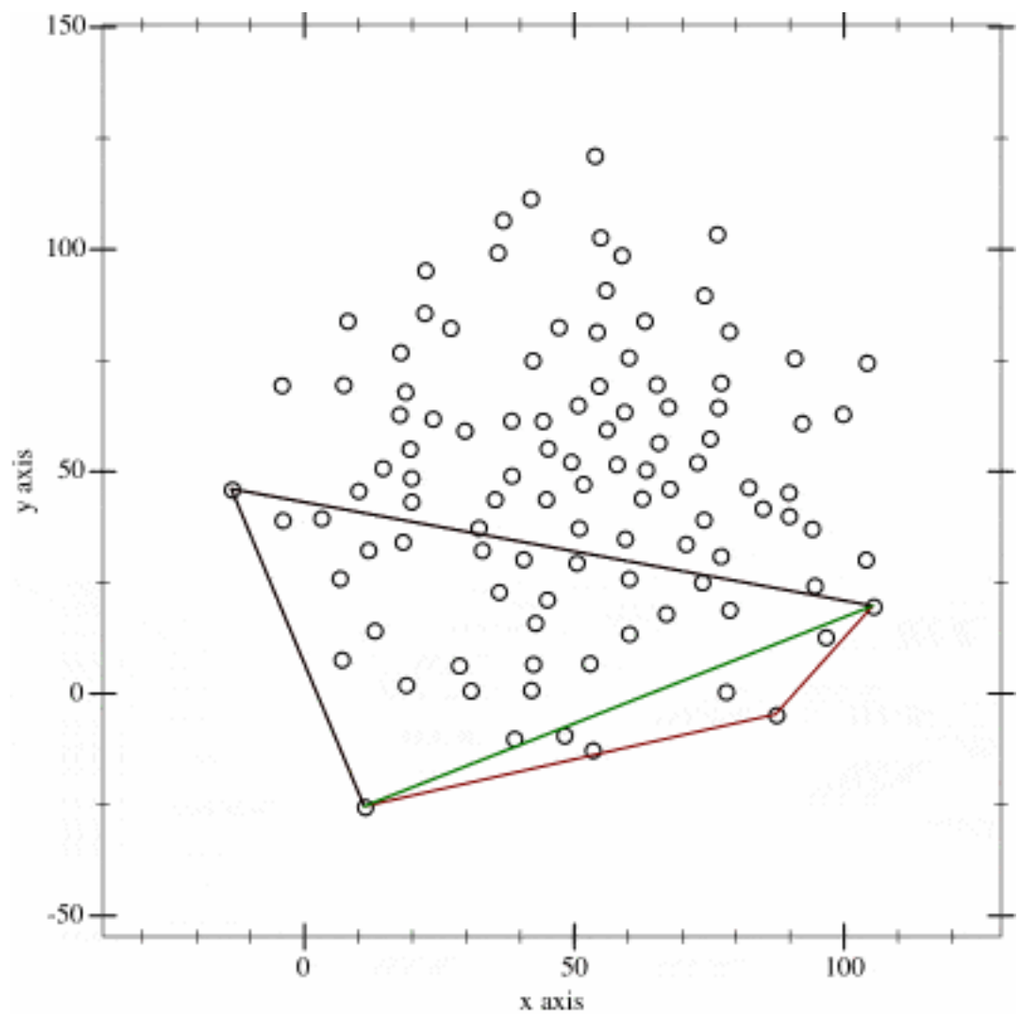
Quick hull example



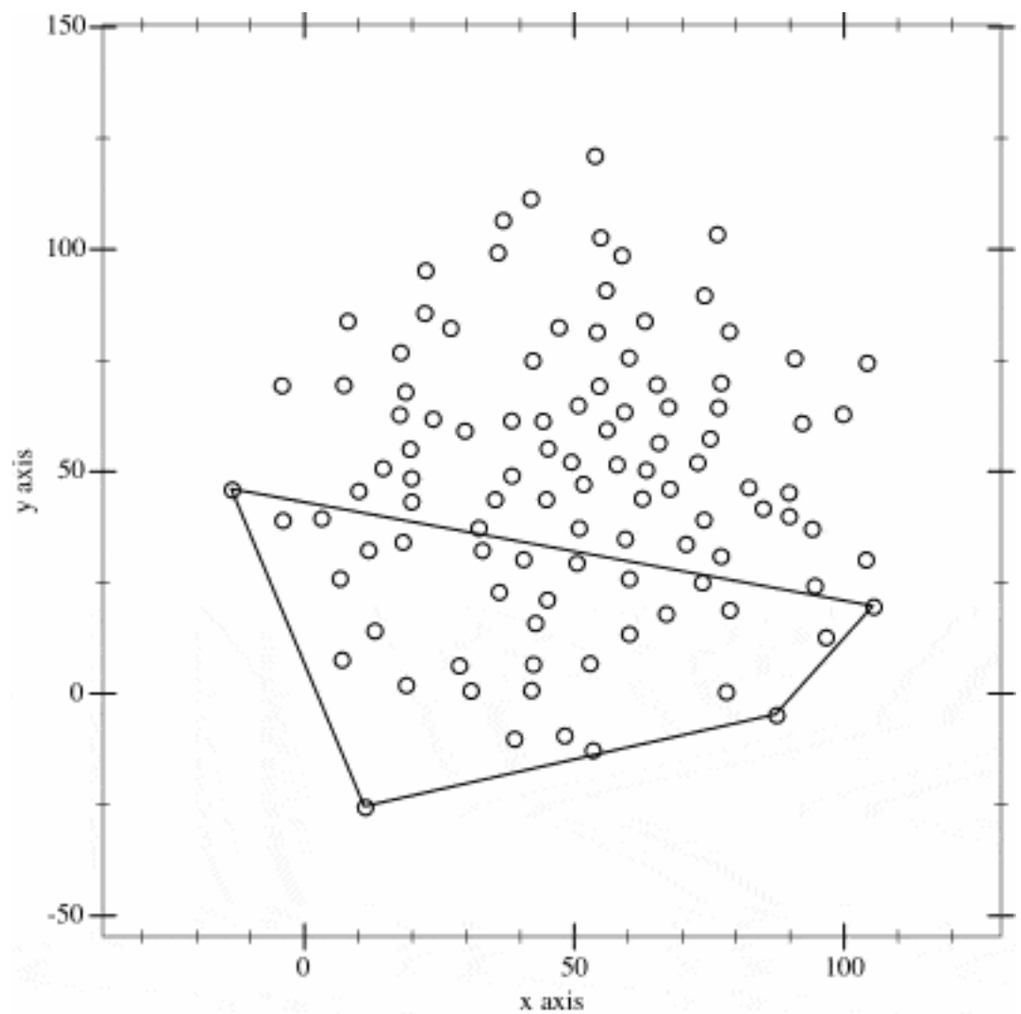
Quick hull example



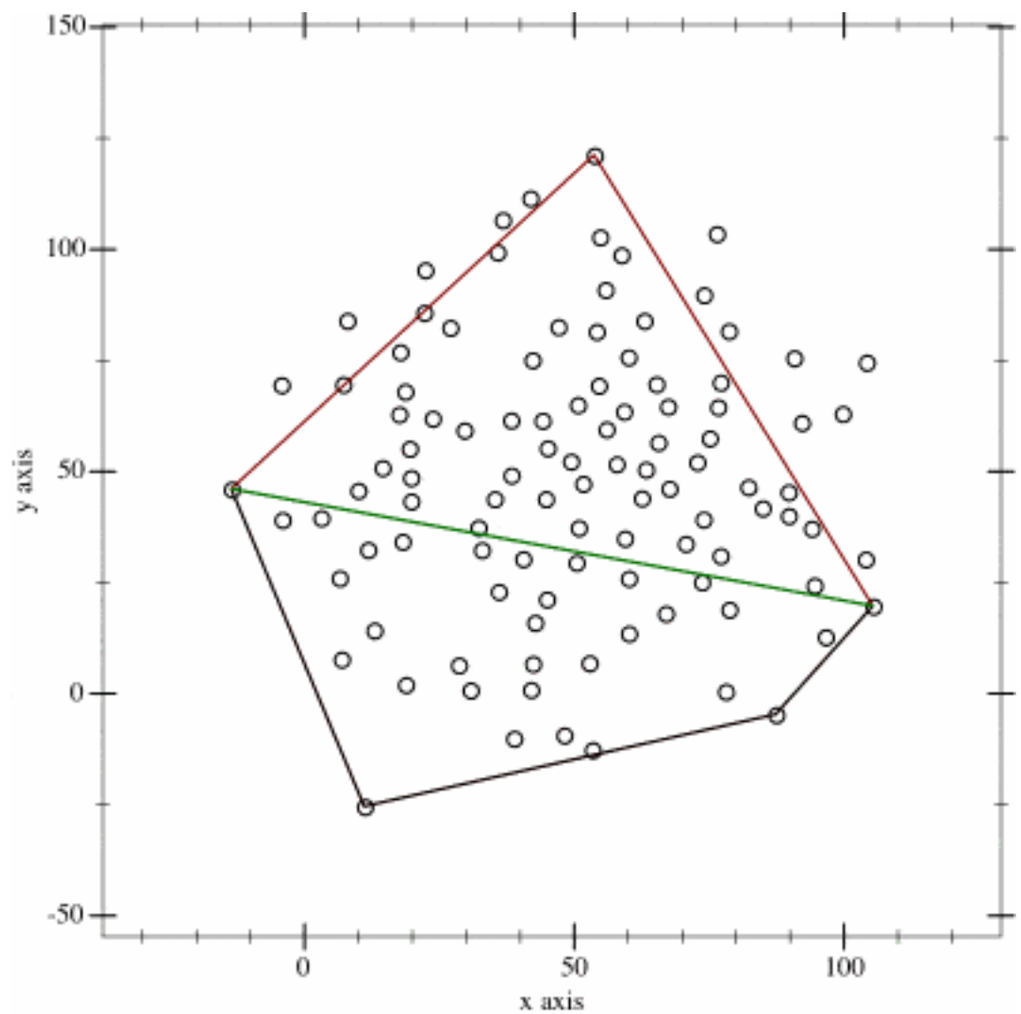
Quick hull example



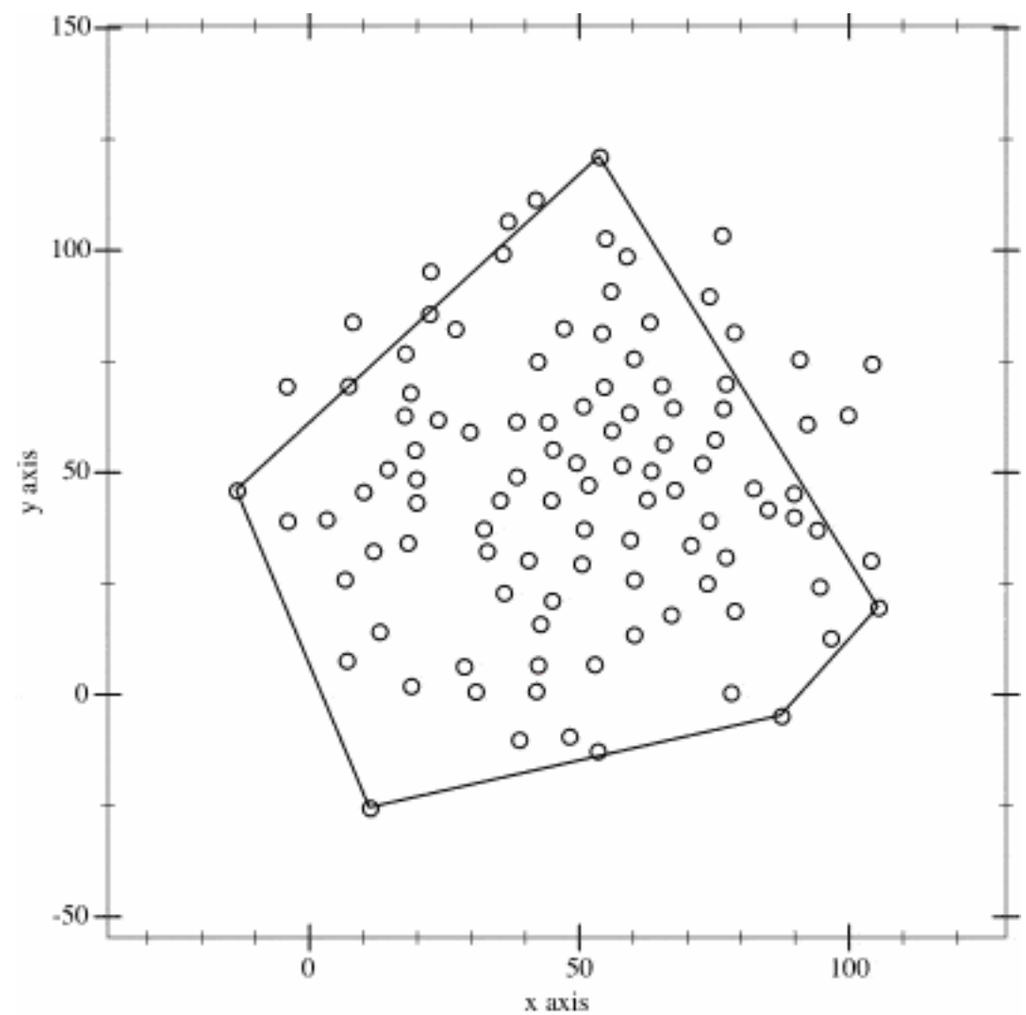
Quick hull example



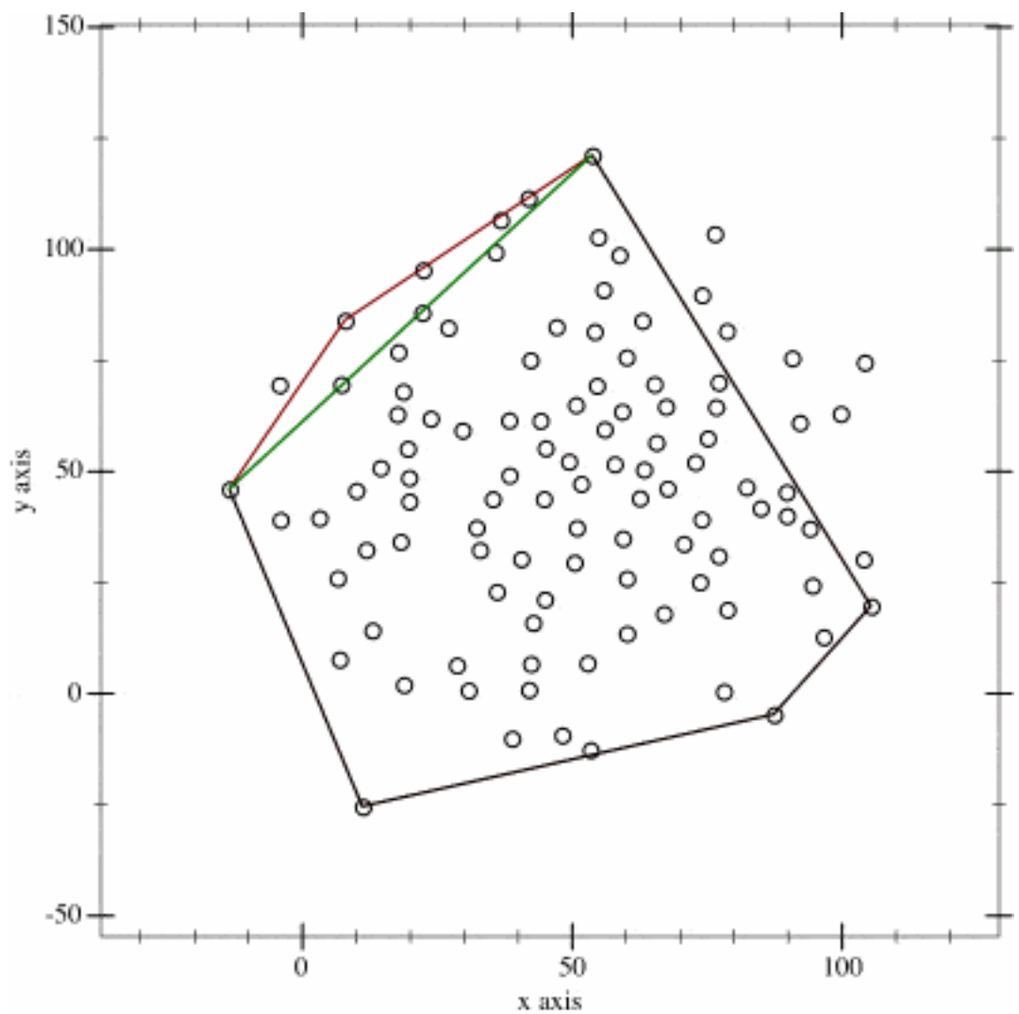
Quick hull example



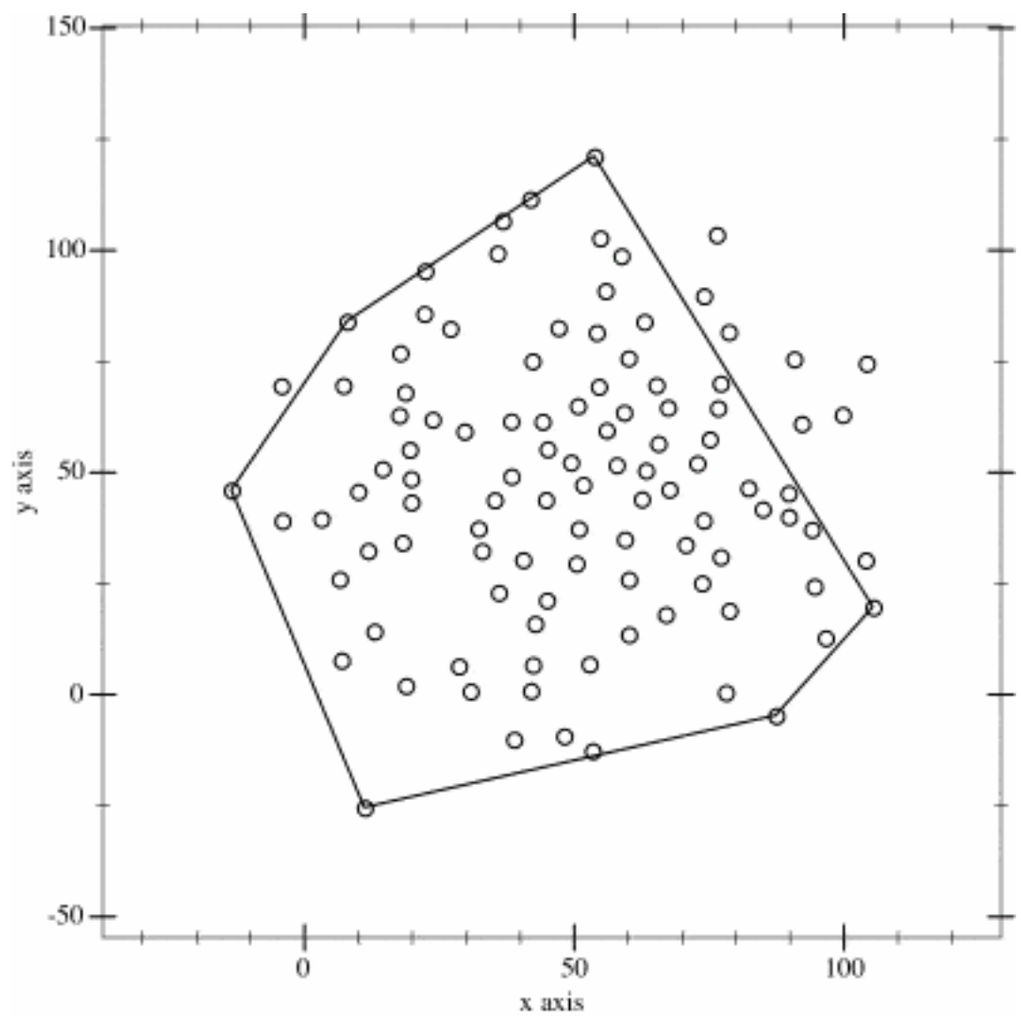
Quick hull example



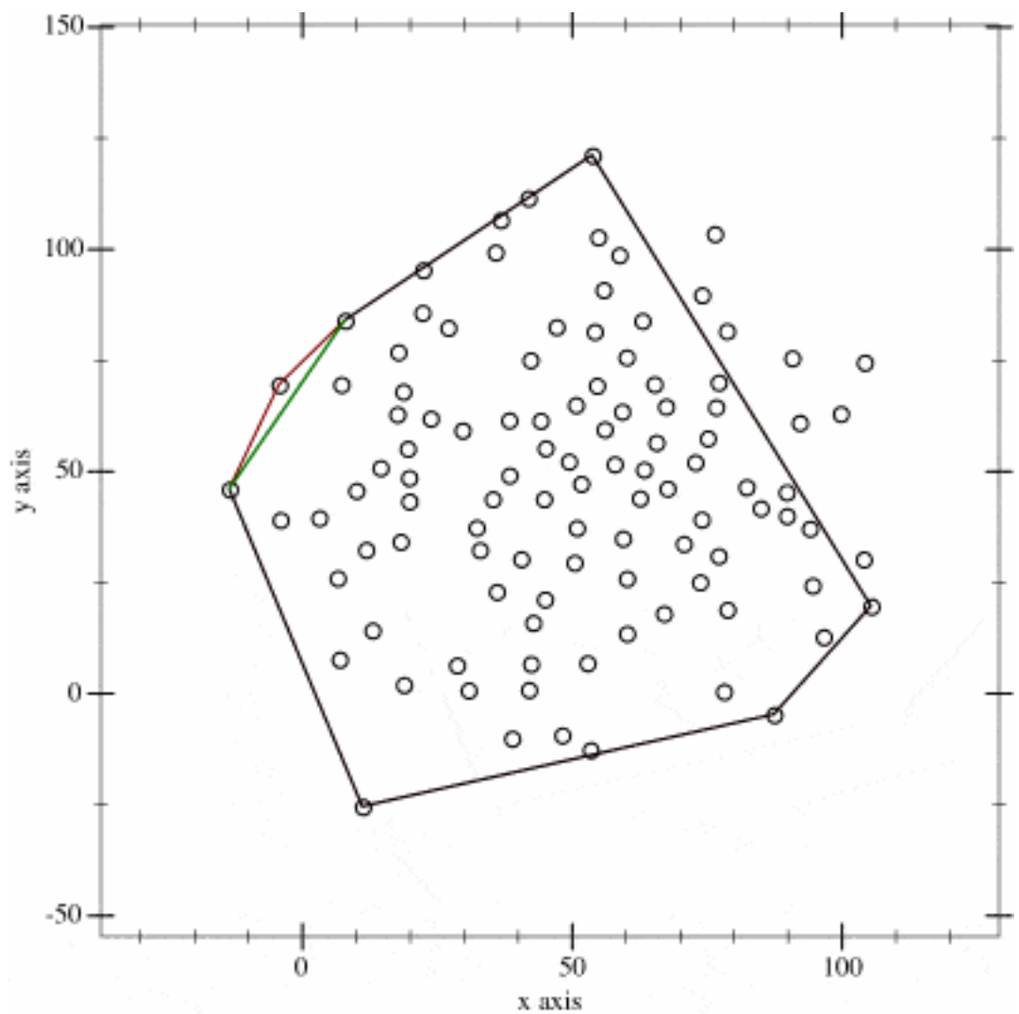
Quick hull example



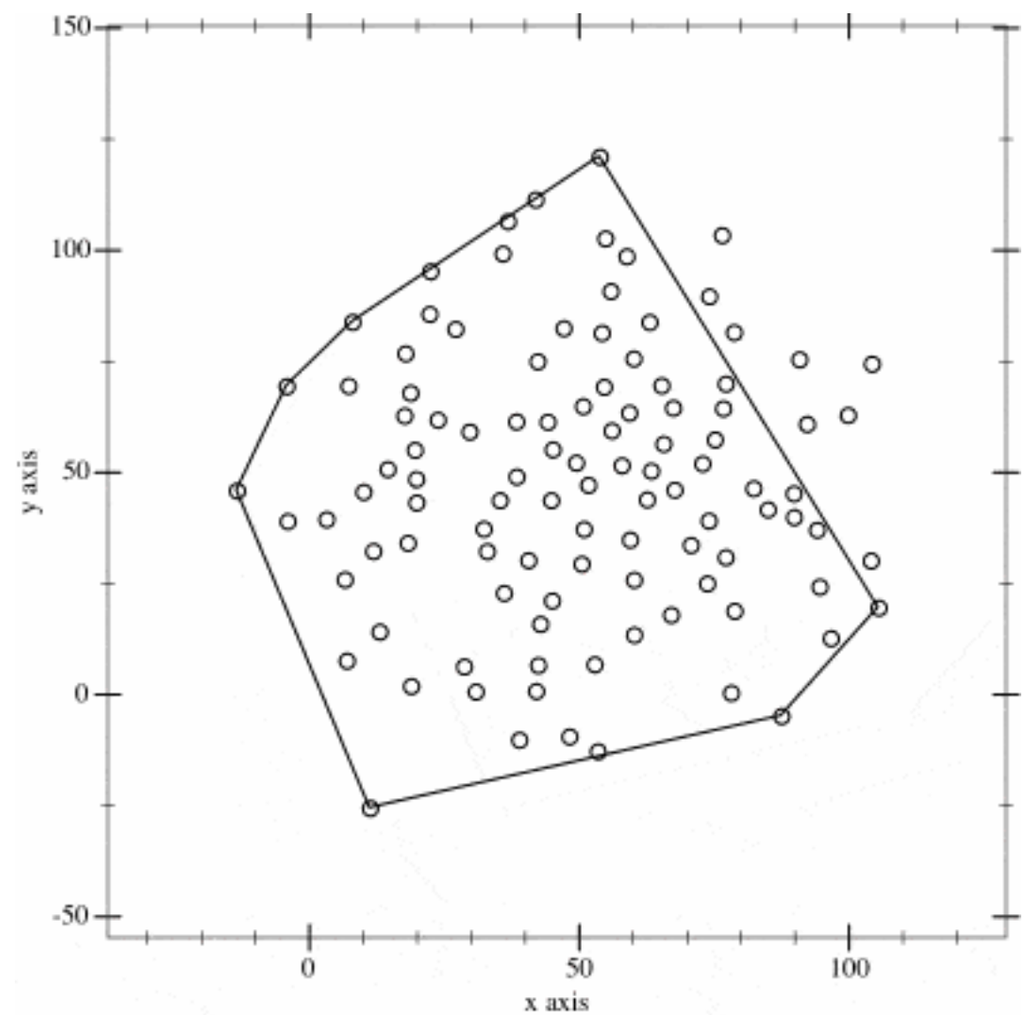
Quick hull example



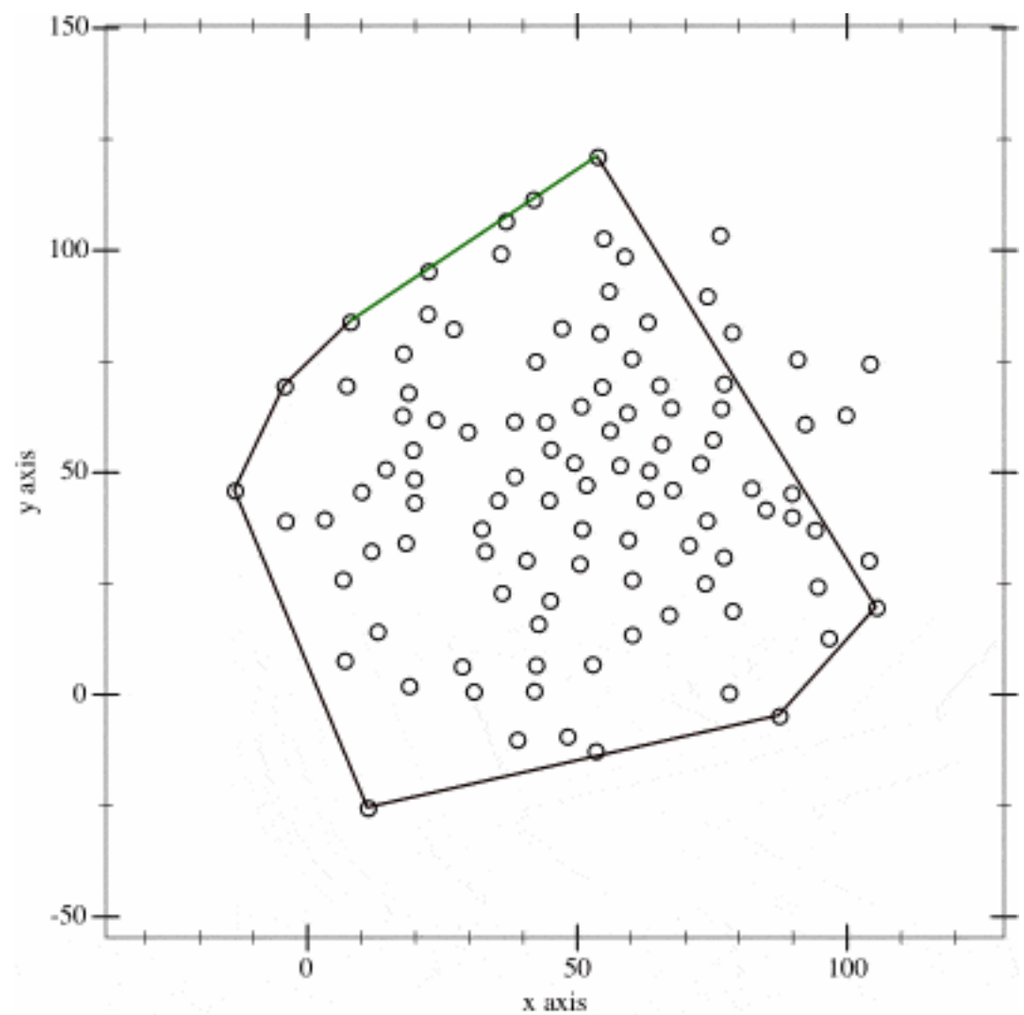
Quick hull example



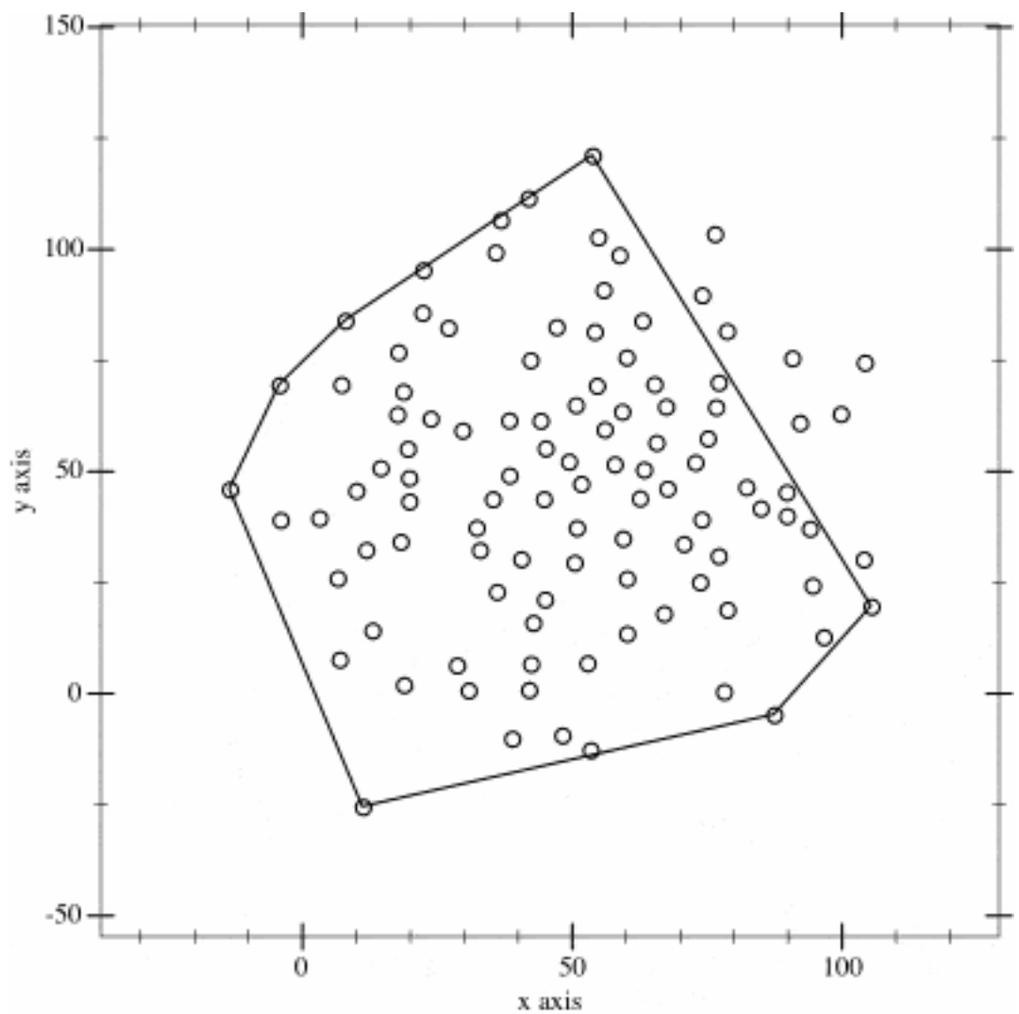
Quick hull example



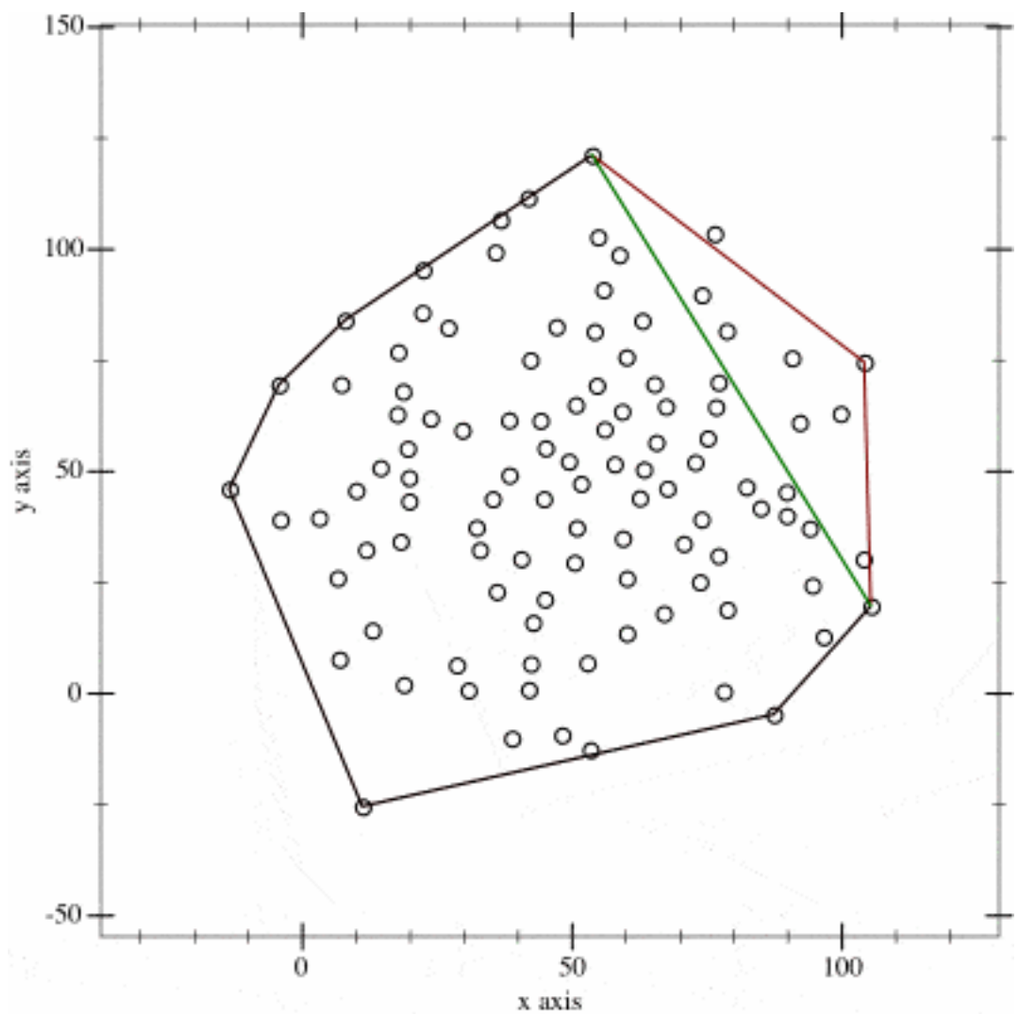
Quick hull example



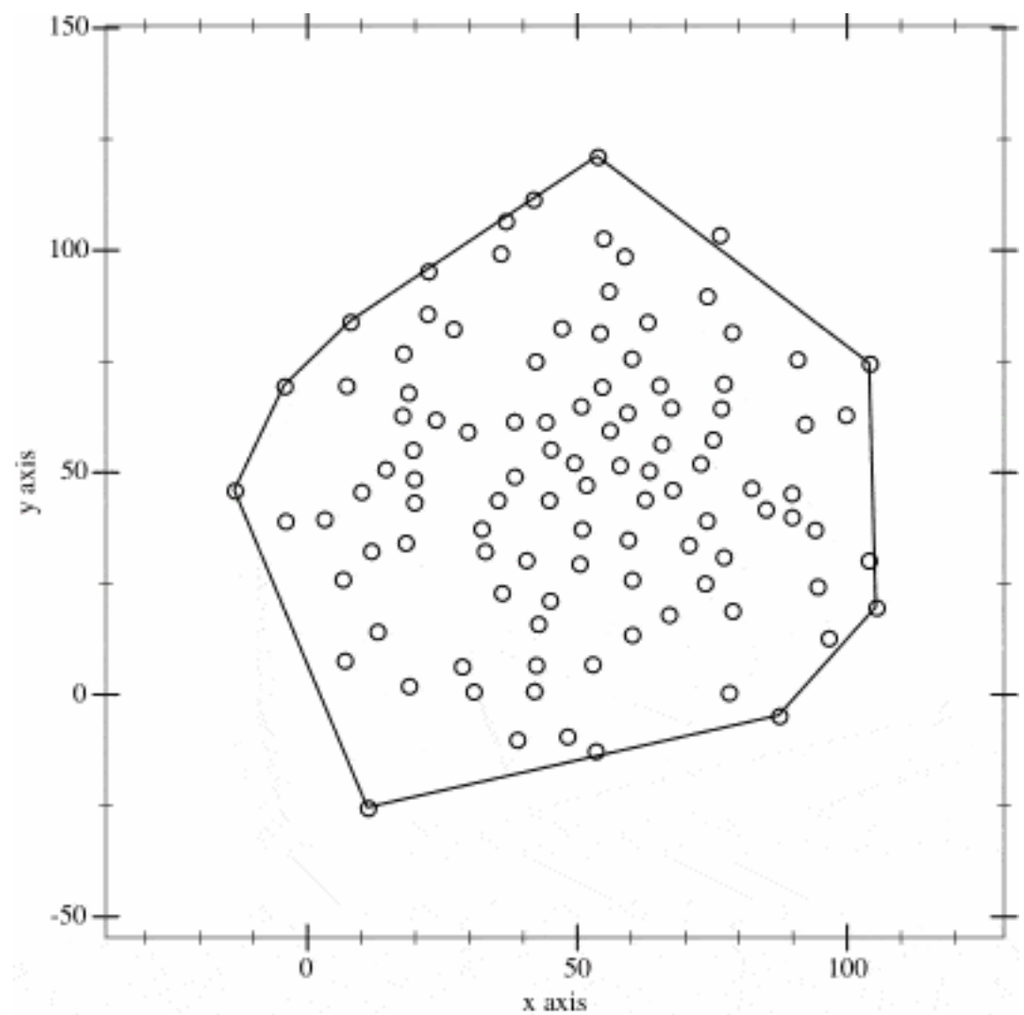
Quick hull example



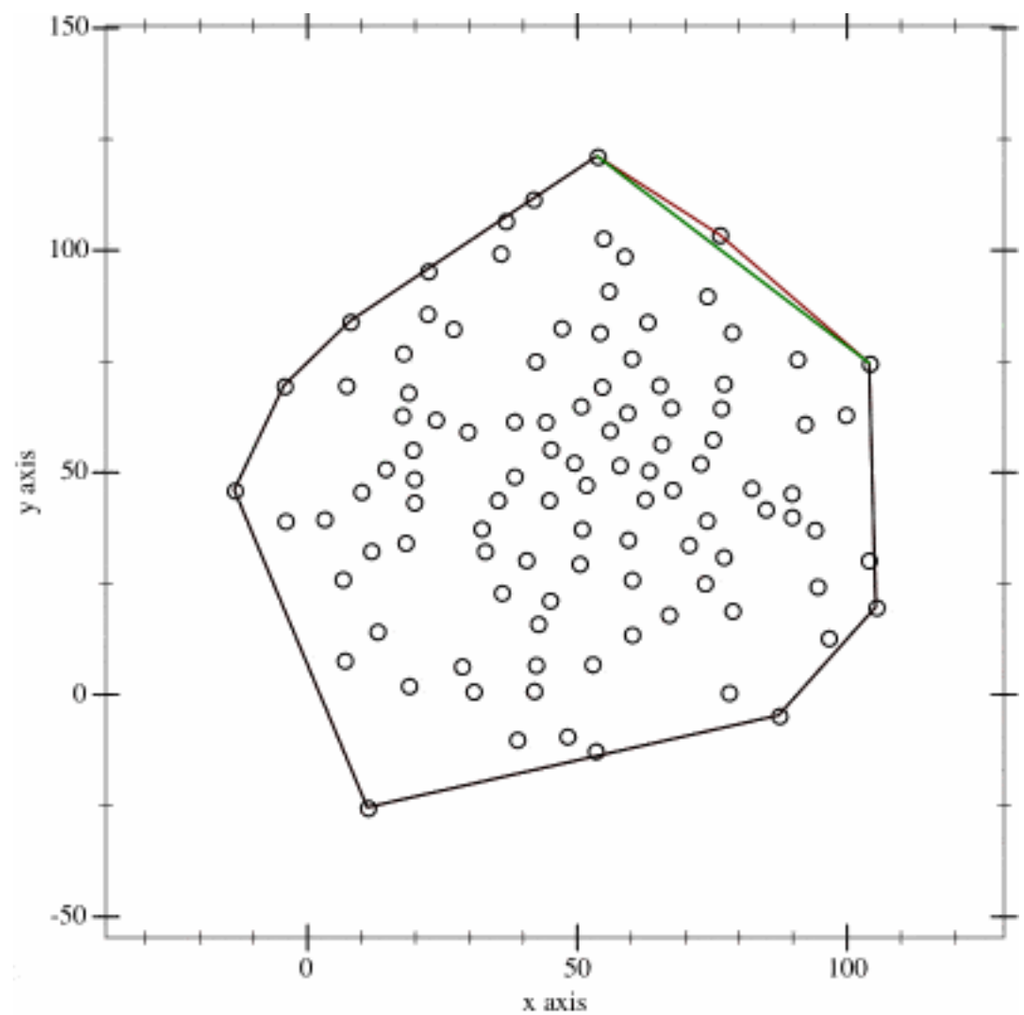
Quick hull example



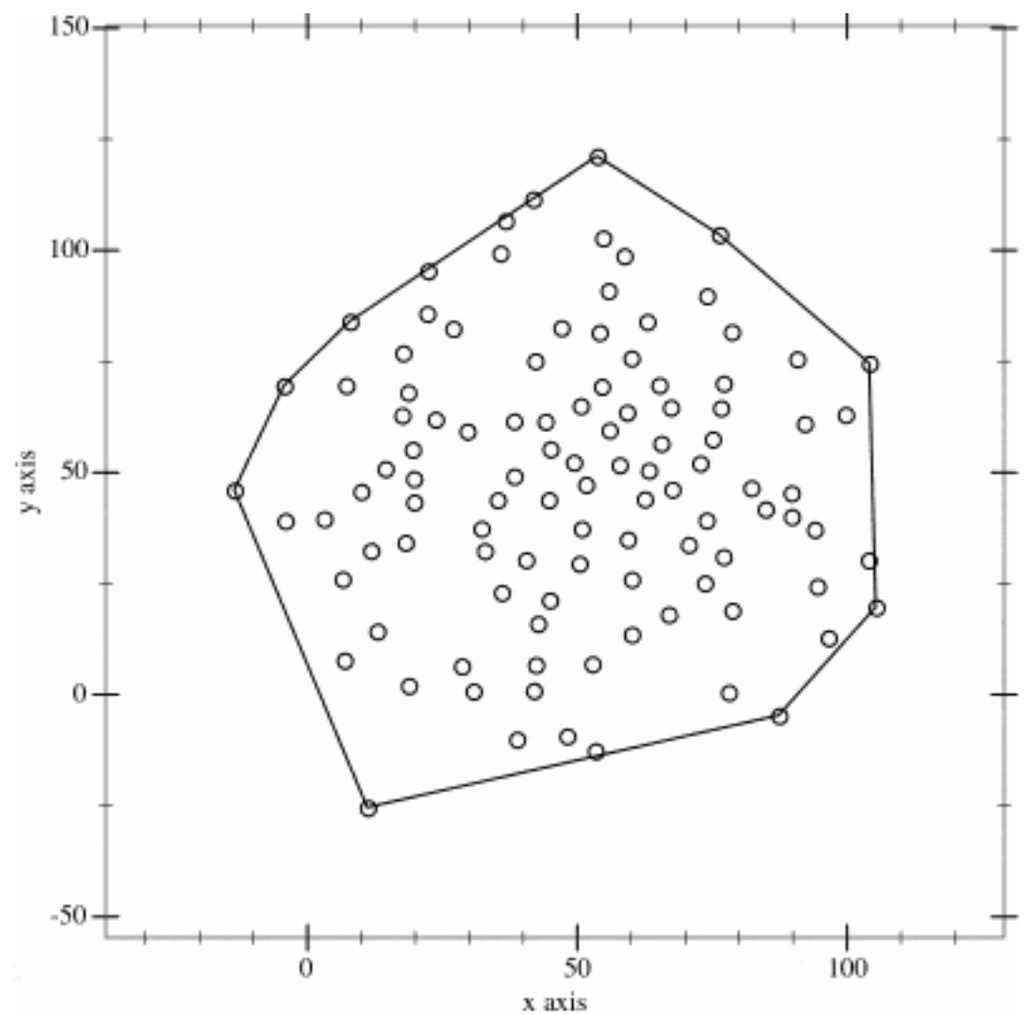
Quick hull example



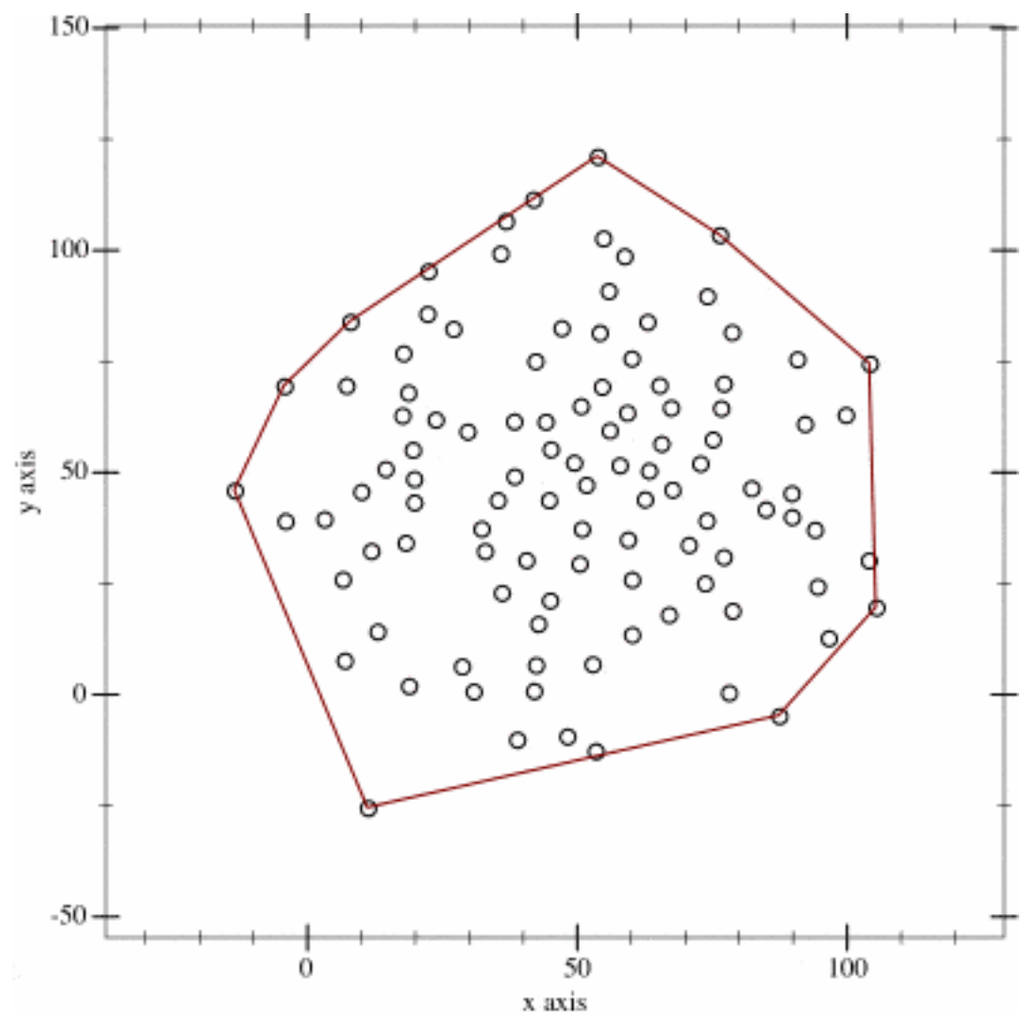
Quick hull example



Quick hull example



Quick hull example



QuickHull algorithm

```
Input = a set S of n points

Assume that there are at least 2 points in the input set S of points

QuickHull (S)
{
    // Find convex hull from the set S of n points
    Convex_Hull := {}

    Find left and right most points, say A & B
    Add A & B to convex hull
    Segment AB divides the remaining (n-2) points into 2 groups S1 and S2
         where S1 are points in S that are on the right side of the oriented line from A to B, and
               S2 are points in S that are on the right side of the oriented line         from B to A

    FindHull (S1, A, B)
    FindHull (S2, B, A)
}
```

QuickHull algorithm

```
FindHull (Sk, P, Q)
{
    // Find points on convex hull from the set Sk of points
    // that are on the right side of the oriented line from P to Q

    If Sk has no point, then return.

    From the given set of points in Sk, find farthest point, say C, from segment PQ

    Add point C to convex hull at the location between P and Q .

    Three points P, Q and C partition the remaining points of Sk into 3 subsets: S0, S1, and S2
           where S0 are points inside triangle PCQ,
                 S1 are points on the right side of the oriented line from  P to C
                 S2 are points on the right side of the oriented line from C to Q

    FindHull(S1, P, C)
    FindHull(S2, C, Q)

}

Output = Convex Hull
```

Smarter way to write

```
QuickHull (S, l, r)

     if S={ }    then return ()
else if S={l, r} then return (l, r)  // a single convex hull edge
else
    z = index of a point that is furthest (max distance) from xy.
    Let A be the set containing points strictly right of (x, z)
    Let B be the set containing points strictly right of (z, y)
    return {QuickHull (A, x, z) U (z) U QuickHull (B, z, y)}
```

Complexity

The partition is determined by the line passing through two distinct extreme points: the rightmost lowest `r` and the leftmost highest points `l`. Finding the extremes require `O(n)` time.

For the recursive function, it takes `n` steps to determine the extreme point `z`, but the cost of recursive calls depends on the sizes of set `A` and set `B`.

Best case. Consider the best possible case, when each partition is almost balanced. Then we have

`T(n) = 2 T(n/2) + O(n)`.

This is a familiar recurrence relation, whose solution is

`T(n) = O(n log(n))`.

This would occur with randomly distributed points.

Worst case. The worst case occurs when each partition is an extremely unbalanced. In that case the recurrence relation is

```
T(n) = T(n-1) + O(n)
     = T(n-1) + cn
```

Repeated expansion shows this is `O(n^2)`. Therefore, in the worst case the QuickHull is quadratic.

Divide and Conquer (Part 1)

Debarka Sengupta

Divide and Conquer

- Reduce problem to one or more sub-problems of the same type
- Typically, each sub-problem is at most a constant fraction of the size of the original problem
- Subproblems typically disjoint
- Often gives significant, usually polynomial, speedup
- Examples: Quick Sort, Mergesort, Binary Search, Strassen's Algorithm, Quicksort. FFT, etc.

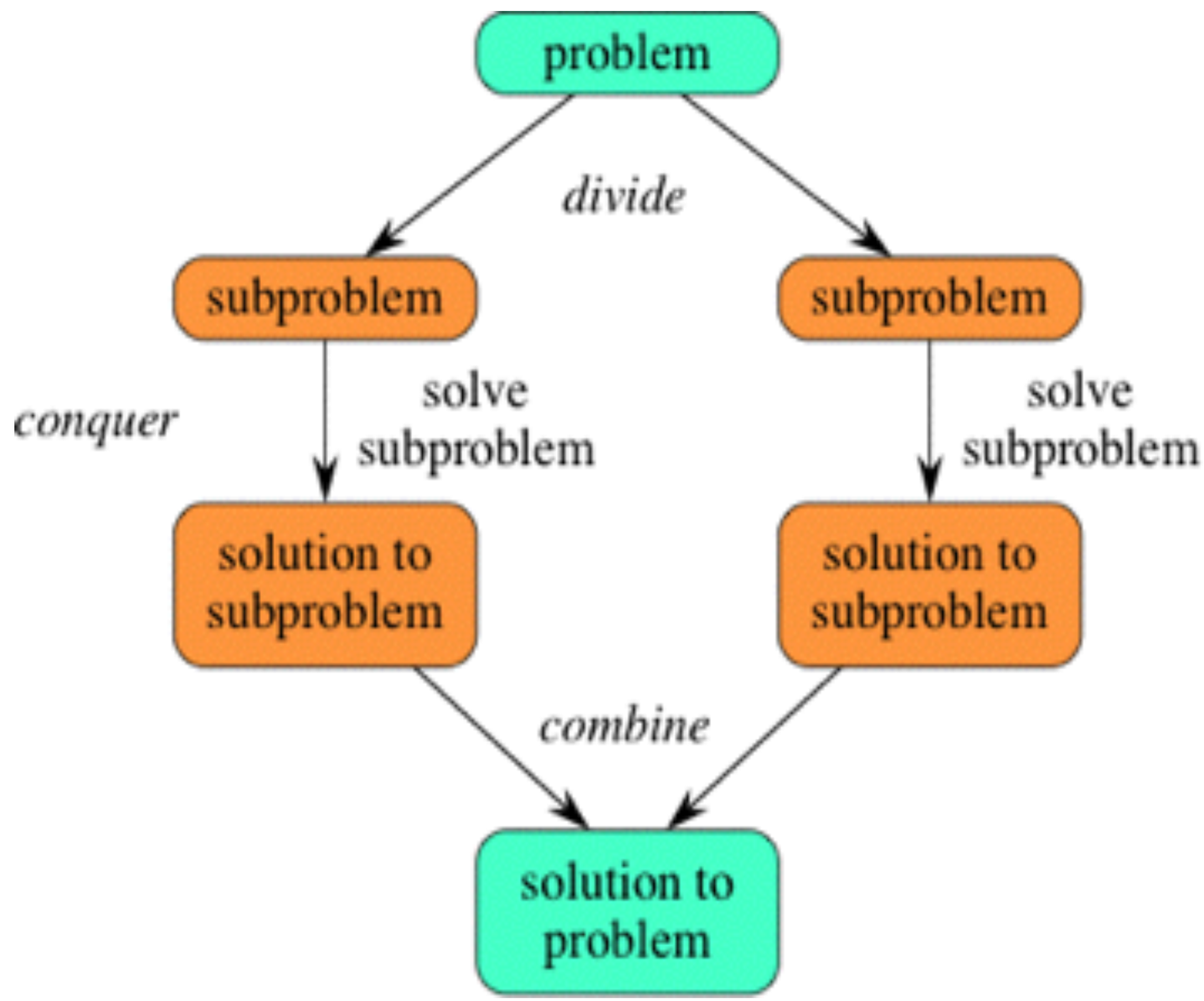
Steps involved

- Divide-and-conquer algorithm has three parts:
- Divide: the problem into a number of subproblems that are smaller instances of the same problem.
- Conquer: the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
- Combine: the solutions to the subproblems into the solution for the original problem.Consequence:

Benefit

- Brute force / naïve solution: N^2 (typically)
- Divide-and-conquer: N log N

Algorithm sketch - two subproblems



Algorithm sketch - more than two subproblems

Finding word in dictionary

Suppose you want to find **“janissary”** in a dictionary:

- open the book near the middle
- the heading on the top left page is **“kiwi”**,
- so move back a small number of pages
- here you find **“hypotenuse”**, so move forward
- find **“ichthyology”**, move forward again

The number of pages you move gets smaller (or at least adjusts in response to the words you find)

Common D&C algorithms

- **Mathematics**
  - Polynomial & Matrix Multiplication
  - Exponentiation
  - Large Integer Manipulation
  - FFT
- **Geometry**
  - Convex Hull
  - Closest Pair
- **Searching**
  - Binary Search
- **Sorting**
  - Merge Sort
  - Quick Sort

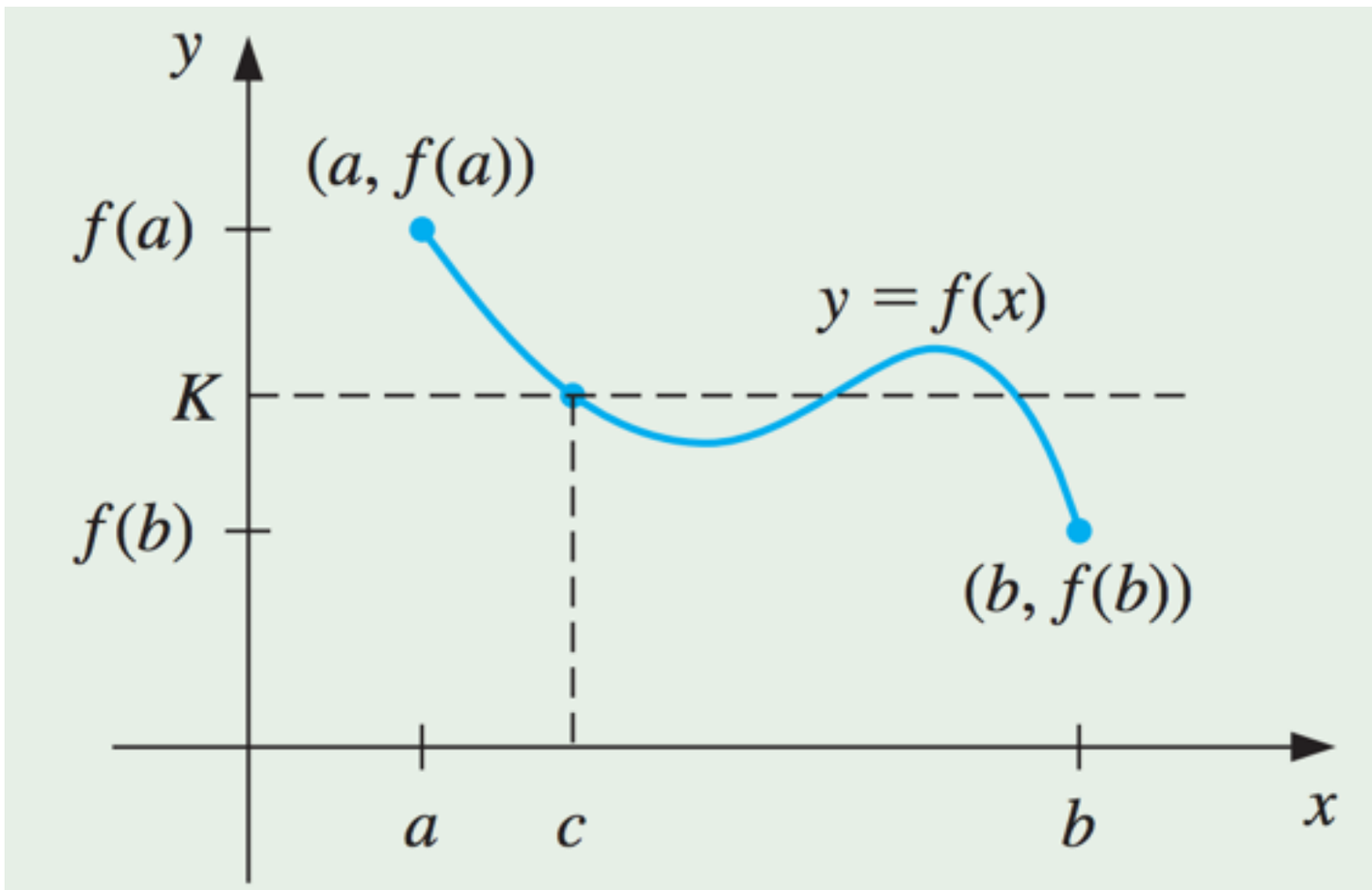
Root finding - bisection

Root finding problem

- Finding Zero of function f(x)
- This process involves finding a root, or solution, of an equation of the form f(x) = 0 for a given function f

Intermediate value theorem (IVT)

It simply states that for any value L between f(a) & f(b), there's a value c in [a,b] for which f(c)=L.



Example

- Show that $f(x) = x^3 + 4x^2 - 10 = 0$ has a root in [1, 2] and use the Bisection method to determine an approximation to the root that is accurate to at least within 10^{-4}

Solution sketch

- Because $f(1) = -5$ and $f(2) = 14$ the IVT ensures that this continuous function has a root in $[1, 2]$.

Solution sketch continued …

- For the first iteration of the Bisection method we use the fact that at the midpoint of [1, 2] we have $f(1.5) = 2.375 > 0$.

- This indicates that we should select the interval [1, 1.5] for our second iteration.

- Then we find that $f(1.25) = -1.796875$ so our new interval becomes [1.25, 1.5], whose midpoint is 1.375.

- Continuing in this manner gives the values shown in the following table

Iterations

Iter	a_n	b_n	p_n	$f(a_n)$	$f(p_n)$	RelErr
1	1.000000	2.000000	1.500000	-5.000	2.375	0.33333
2	1.000000	1.500000	1.250000	-5.000	-1.797	0.20000
3	1.250000	1.500000	1.375000	-1.797	0.162	0.09091
4	1.250000	1.375000	1.312500	-1.797	-0.848	0.04762
5	1.312500	1.375000	1.343750	-0.848	-0.351	0.02326
6	1.343750	1.375000	1.359375	-0.351	-0.096	0.01149
7	1.359375	1.375000	1.367188	-0.096	0.032	0.00571
8	1.359375	1.367188	1.363281	-0.096	-0.032	0.00287
9	1.363281	1.367188	1.365234	-0.032	0.000	0.00143
10	1.363281	1.365234	1.364258	-0.032	-0.016	0.00072
11	1.364258	1.365234	1.364746	-0.016	-0.008	0.00036
12	1.364746	1.365234	1.364990	-0.008	-0.004	0.00018
13	1.364990	1.365234	1.365112	-0.004	-0.002	0.00009

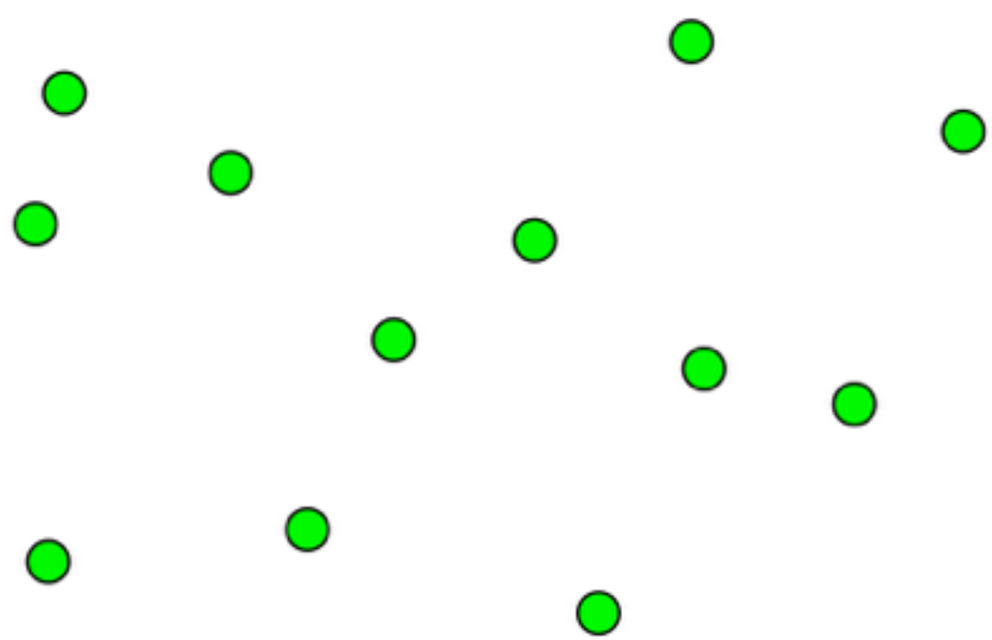
Precision

- The number of iterations depends on the precision being looked for
- if $c_1 = (a+b)/2$ is the midpoint of the initial interval, and c_n is the midpoint of the interval in the nth step, then the difference between c_n and a solution c is bounded by $|c_n - c| <= |b - a|/2^n$
- One should be able to perform the back calculation

Closest Pair

Closest pair - problem statement

- **Given a set of points $\{p_1, \ldots, p_n\}$ find the pair of points $\{p_i, p_j\}$ that are closest**

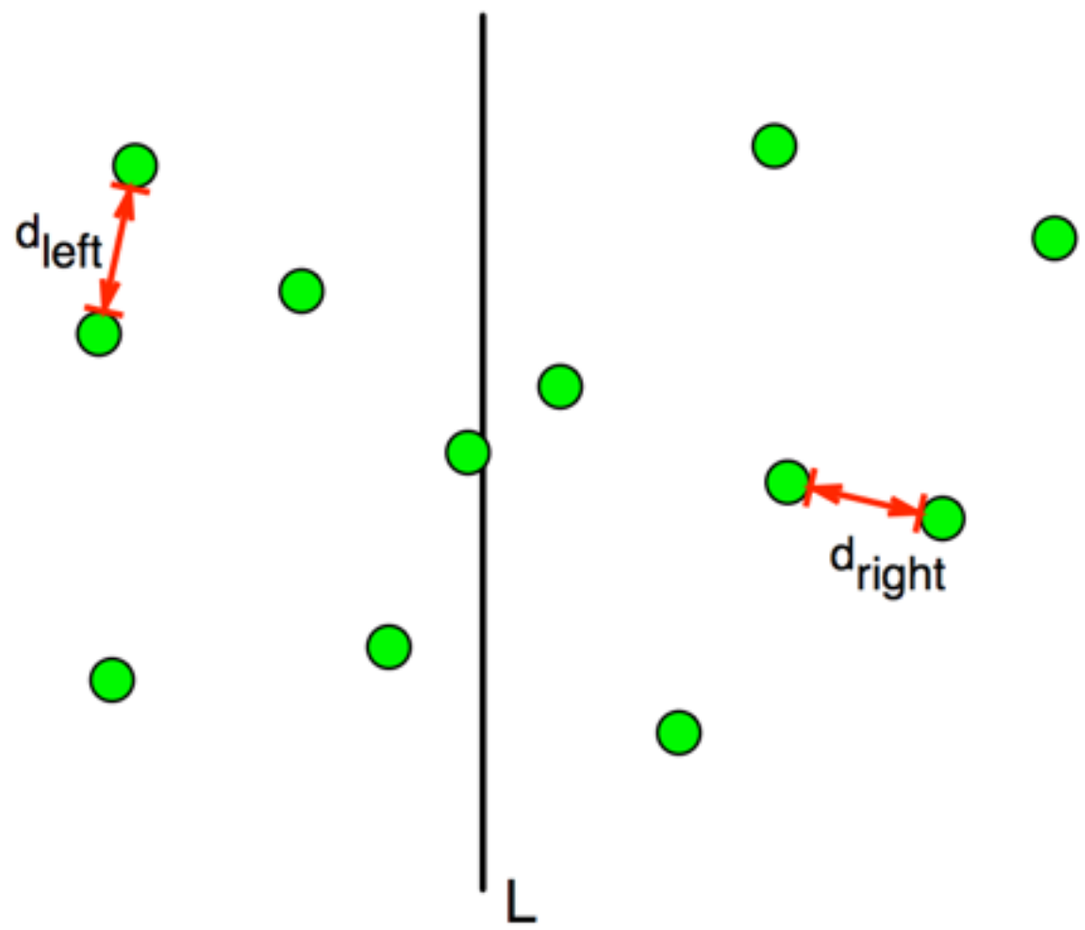


Brute force vs. divide and conquer

- Brute force gives an $O(n^2)$ algorithm to just check ever pair of points.
- Can we do it faster? Seems like difficult without checking every pair.
- In fact, we can find the closest pair in $O(n \log n)$ time.

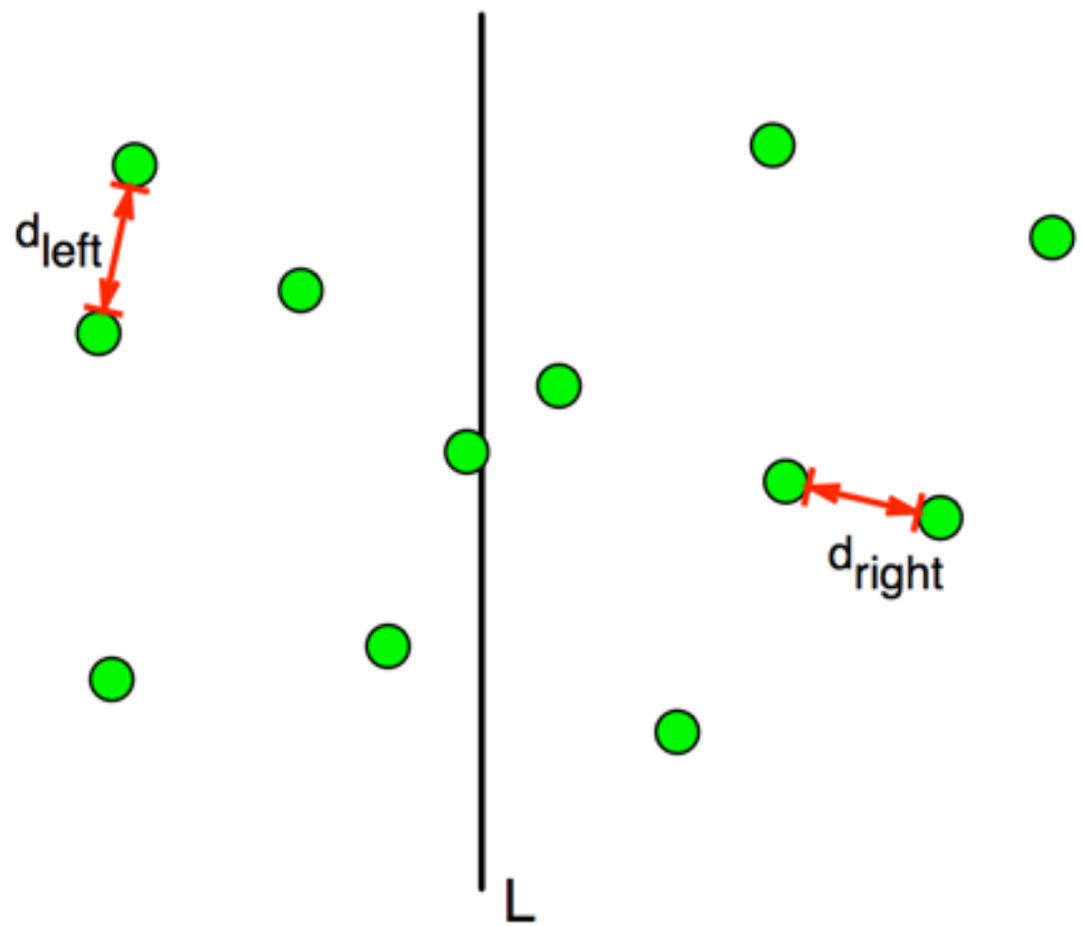
Divide

- **Split the points with line L so that half the points are on each side.**
- **Recursively find the pair of points closest in each half.**



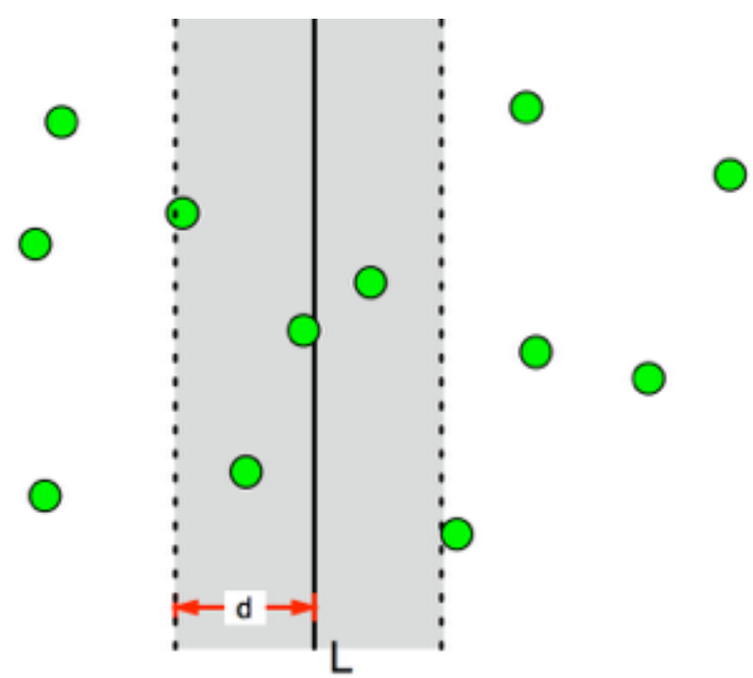
Merge

- **Let $d = \min\{d_{left}, d_{right}\}$**
- **d would be the answer, except maybe L split a close pair!**



Region near L

- **If the closest pair exists across the L, it should contained within d margins on both sides**



A life saver observation

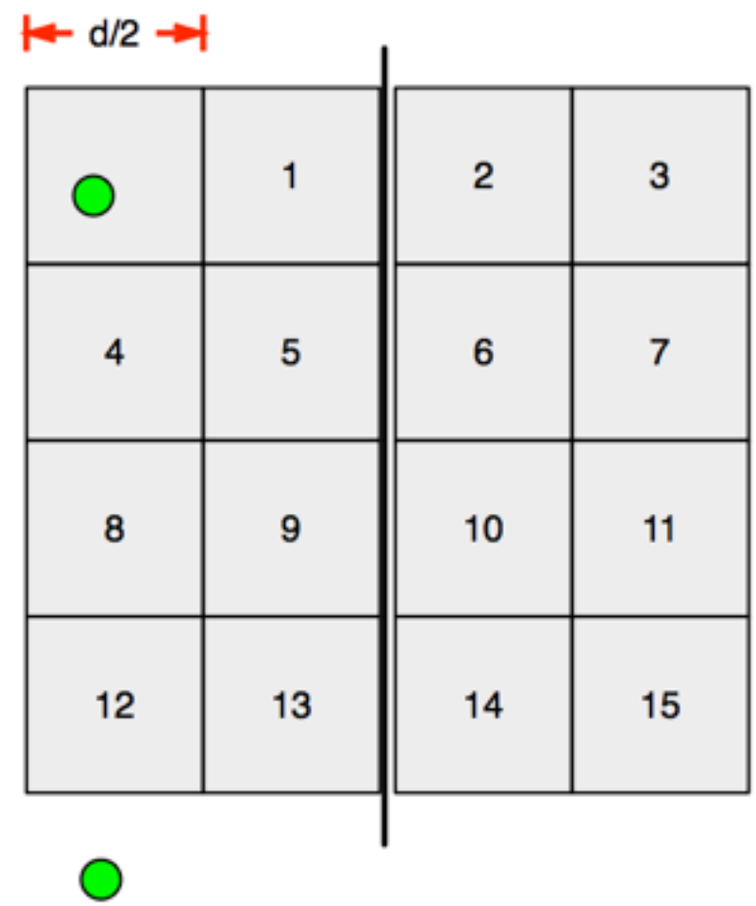
- Let S_y be an array of the points in that region, sorted by decreasing y-coordinate value.
- S_y might contain all the points, so we can't just check every pair inside it.

Theorem

Suppose $S_y = p_1, \ldots, p_m$. *If* $dist(p_i, p_j) < d$ *then* $j - i \leq 15$.

Constant number of checks for each point is enough

- We can split the entire margin area into d/2 sided square boxes
- Each box can have max 1 point
- Suppose 2 points are separated by > 15 indices.
  - Then, at least 3 full rows separate them (the packing shown is the smallest possible).
  - But the height of 3 rows is > 3d/2, which is > d.
  - So the two points are further than d apart.



Conclusion

- Starting from the bottom most point, for each point, it is enough to check next 15 data points up in the sequence of increasing value in S_y
- Constant (15) time for each of the n points

The algorithm

```
ClosestPair(Px, Py):
   if |Px| == 2: return dist(Px[1],Px[2])     // base

   d1 = ClosestPair(FirstHalf(Px,Py))      // divide
   d2 = ClosestPair(SecondHalf(Px,Py))
   d = min(d1,d2)

   Sy = points in Py within d of L      // merge
   For i = 1,...,|Sy|:
       For j = 1,...,15:
           d = min( dist(Sy[i], Sy[j]), d )
   Return d
```

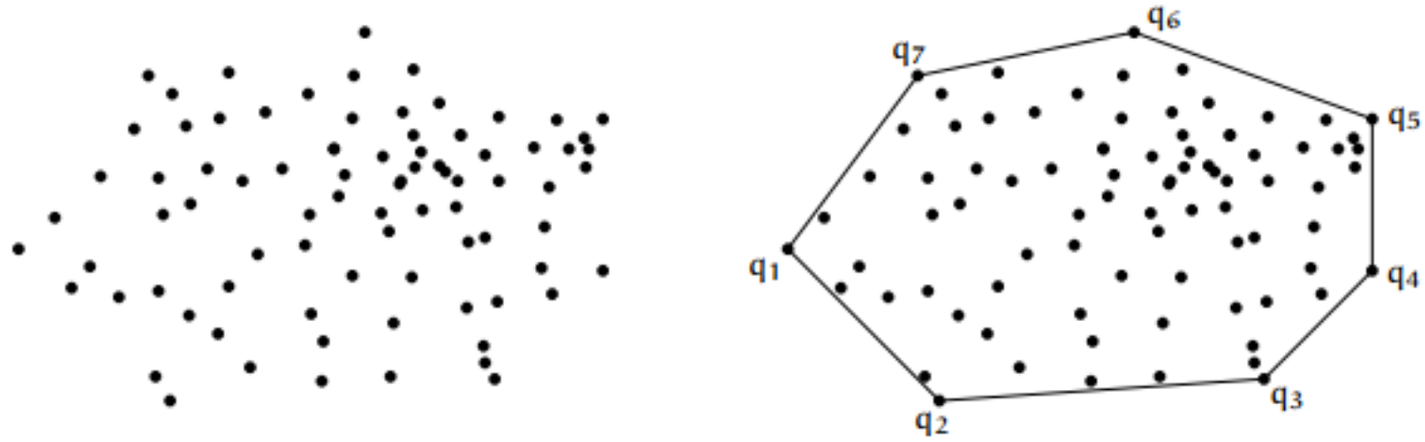
Running time

- Divide set of points in half each time: O(log n) depth recursion
- Merge takes O(n) time.
- Recurrence: $T(n) = 2T(n/2) + cn$
- Analysis is same as MergeSort ⇒ O(n log n) time

Convex Hull

Convex Hull

- Studied in the field of Computational Geometry.
- A convex hull, of a set of N points is the smallest perimeter fence enclosing all these points.
- Convex Hull Output - Sequence of vertices in counterclockwise/ clockwise order.



(a) Input. (b) Output.

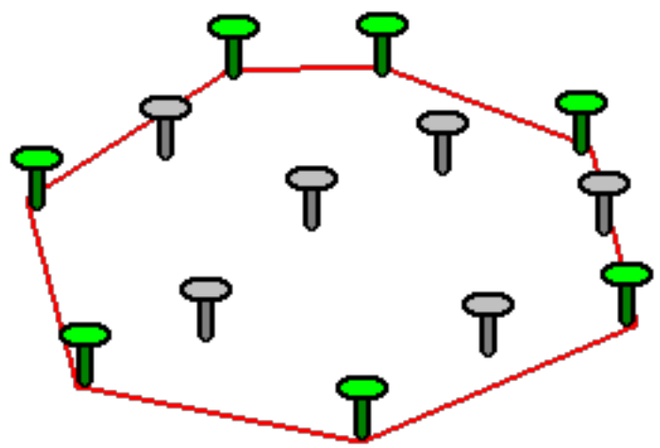
Motion planning

- Robot Motion Planning - Find the shortest path in the plane from a starting point s to an ending point t, that avoids a polygonal obstacle.

- Here, we can see that the shortest path is either the straight line from s to t, or one of the two polygon chains of the convex hull.

Mechanical algorithm

Hammer nails perpendicular to the plane and stretch elastic rubber band around these points.



Brute force

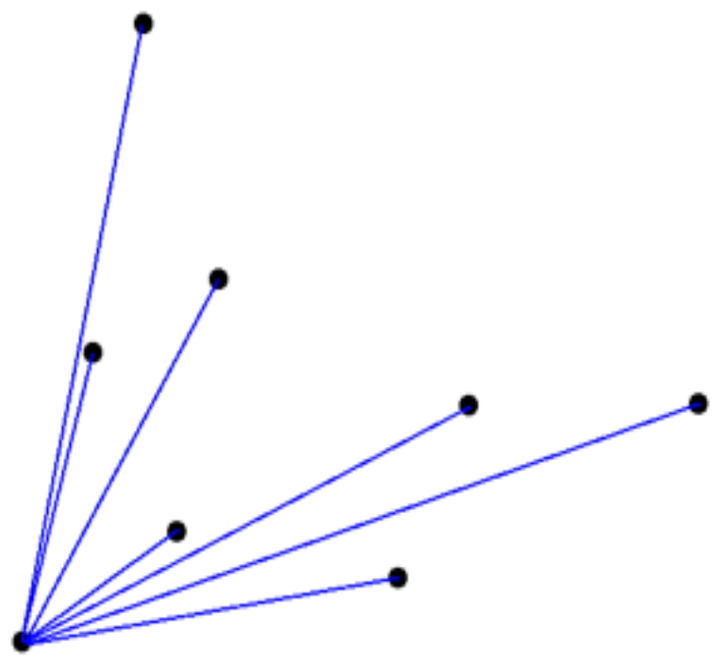
- Create a list of all possible line segments
  - N choose 2 line segments for N points
- For each line segment check if both sides have points
  - If that’s not the case, include the segment in the solution set
- Quadratic complexity algorithm

Graham's scan

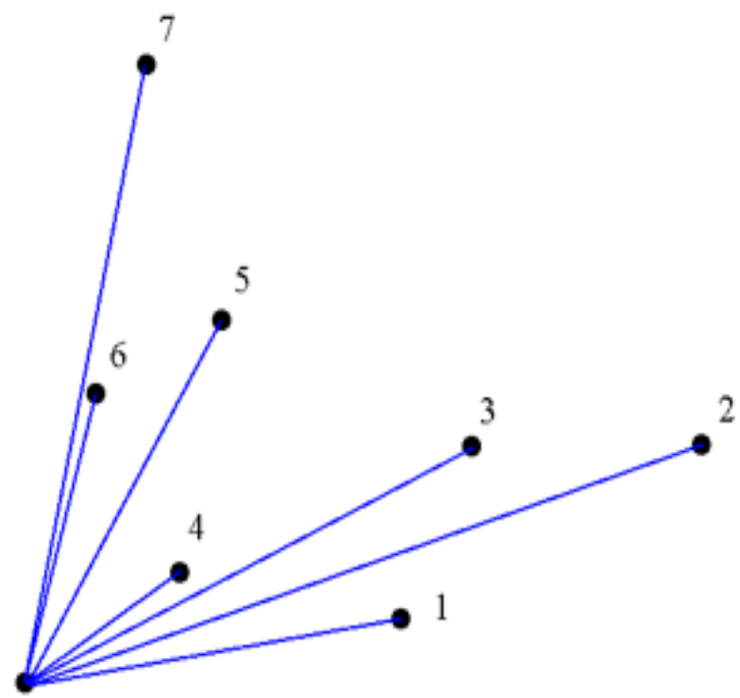
- Start at point guaranteed to be on the hull. (the point with the minimum y value)
- Sort remaining points by polar angles of vertices relative to the first point.
- Go through sorted points, keeping vertices of points that have left turns and dropping points that have right turns.

Graham’s scan example

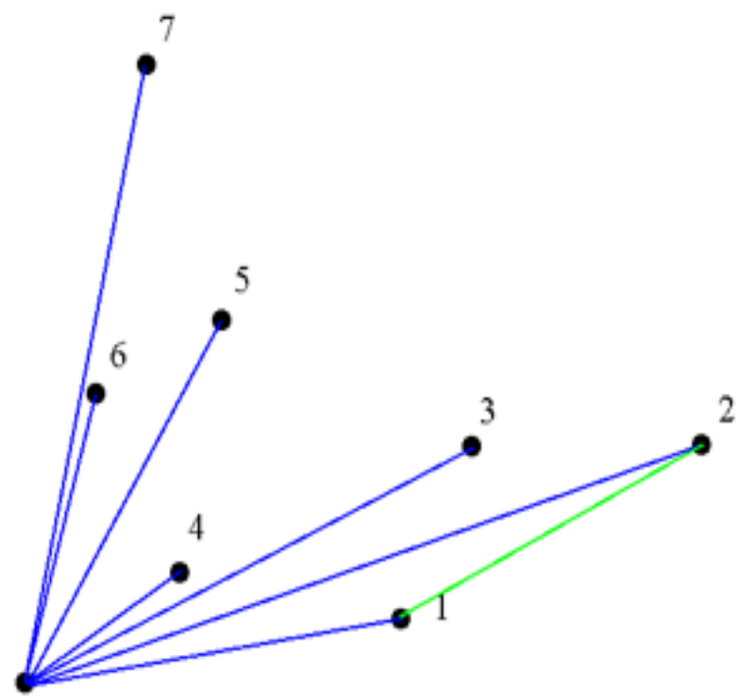
Graham’s scan example



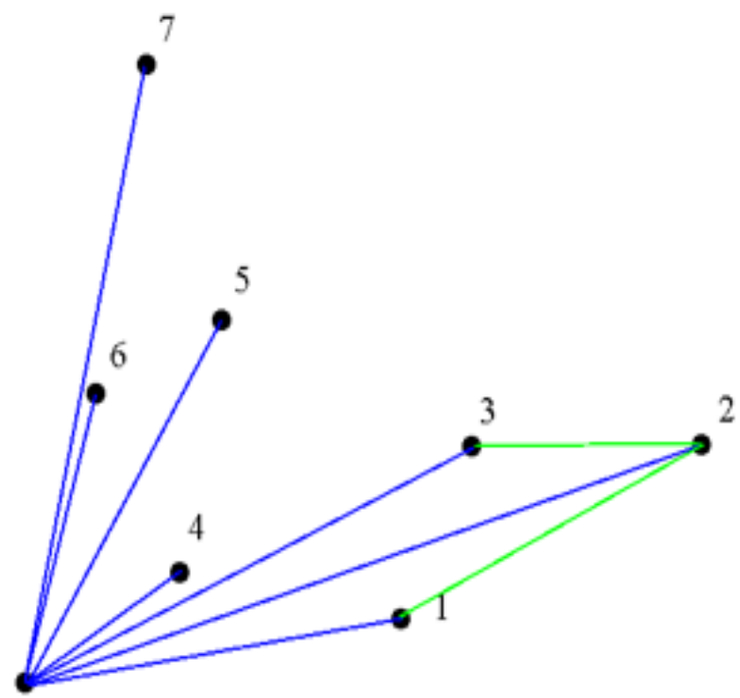
Graham’s scan example



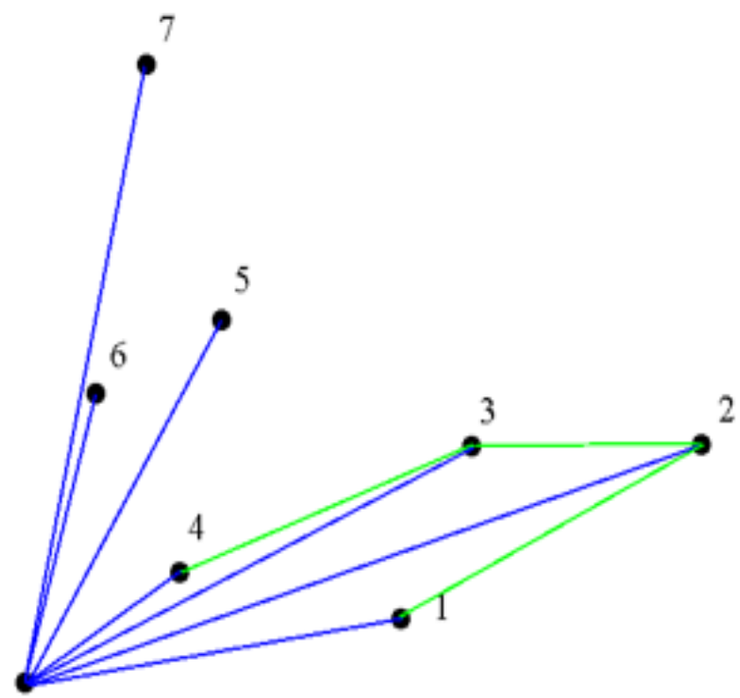
Graham’s scan example



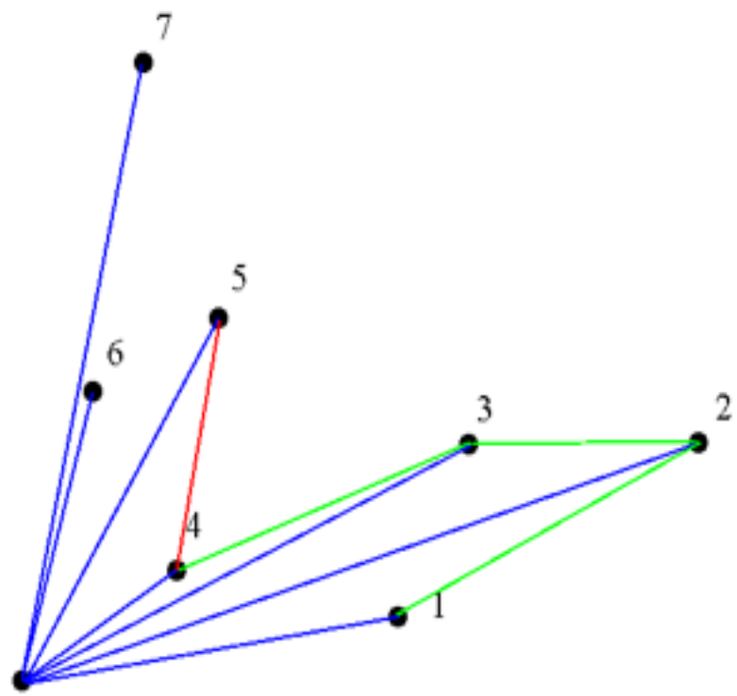
Graham’s scan example



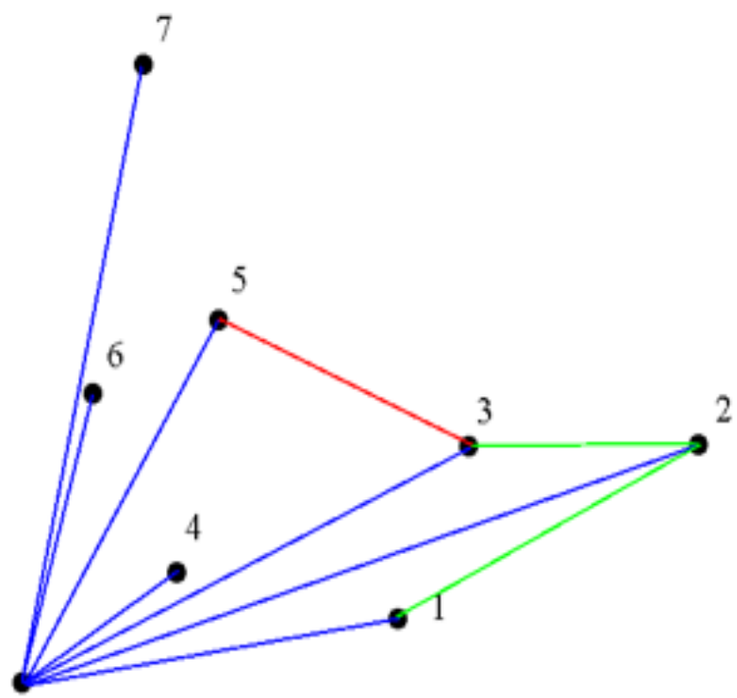
Graham's scan example



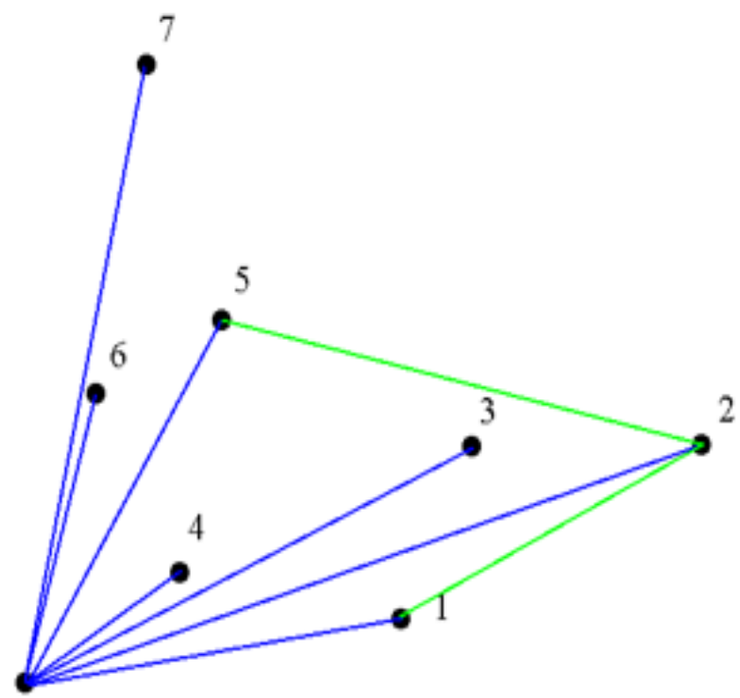
Graham's scan example



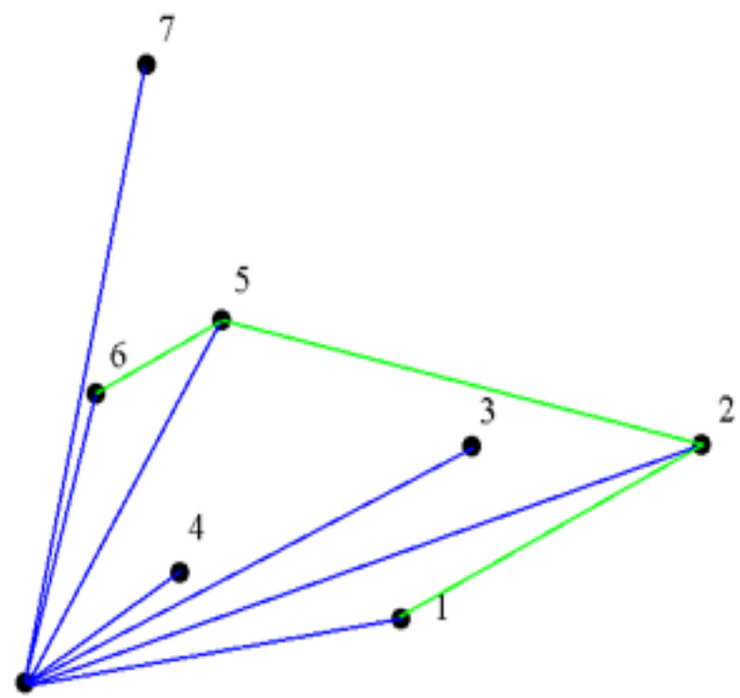
Graham’s scan example



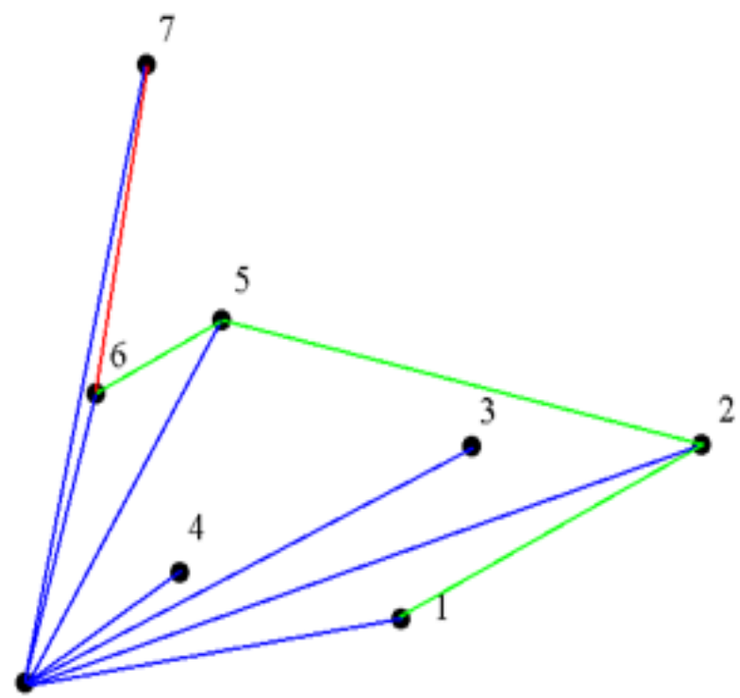
Graham's scan example



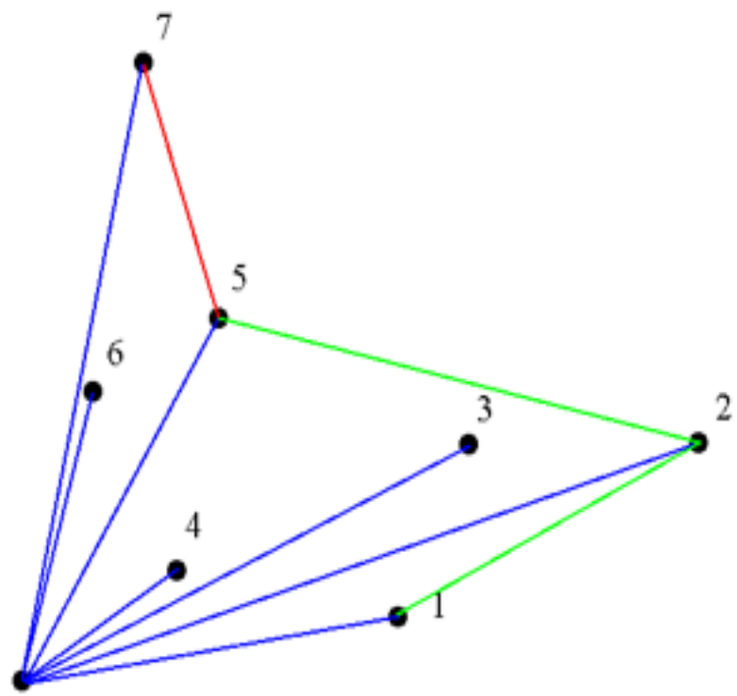
Graham's scan example



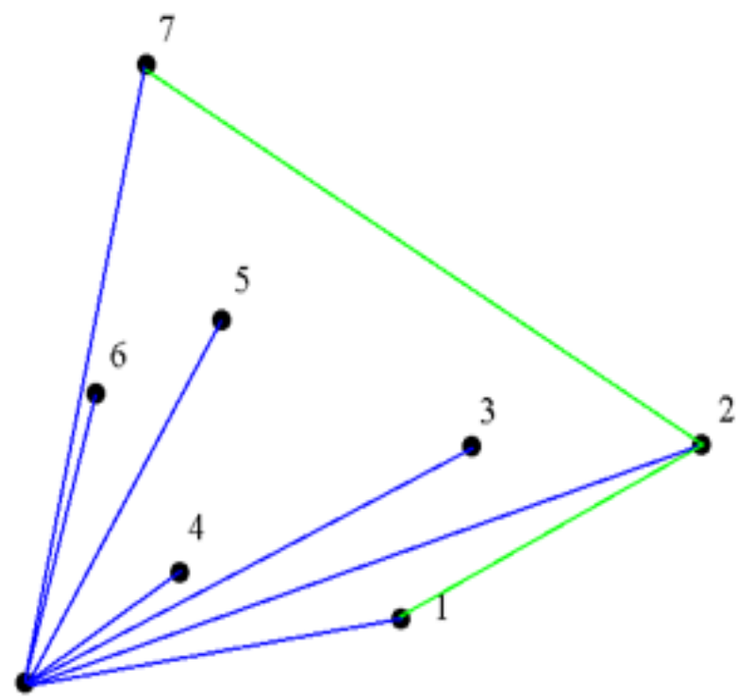
Graham’s scan example



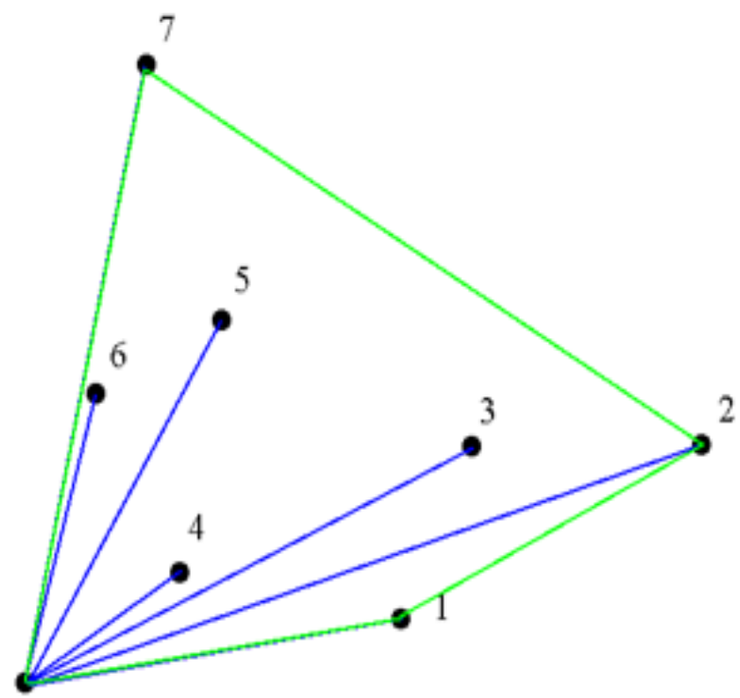
Graham’s scan example



Graham’s scan example



Graham's scan example

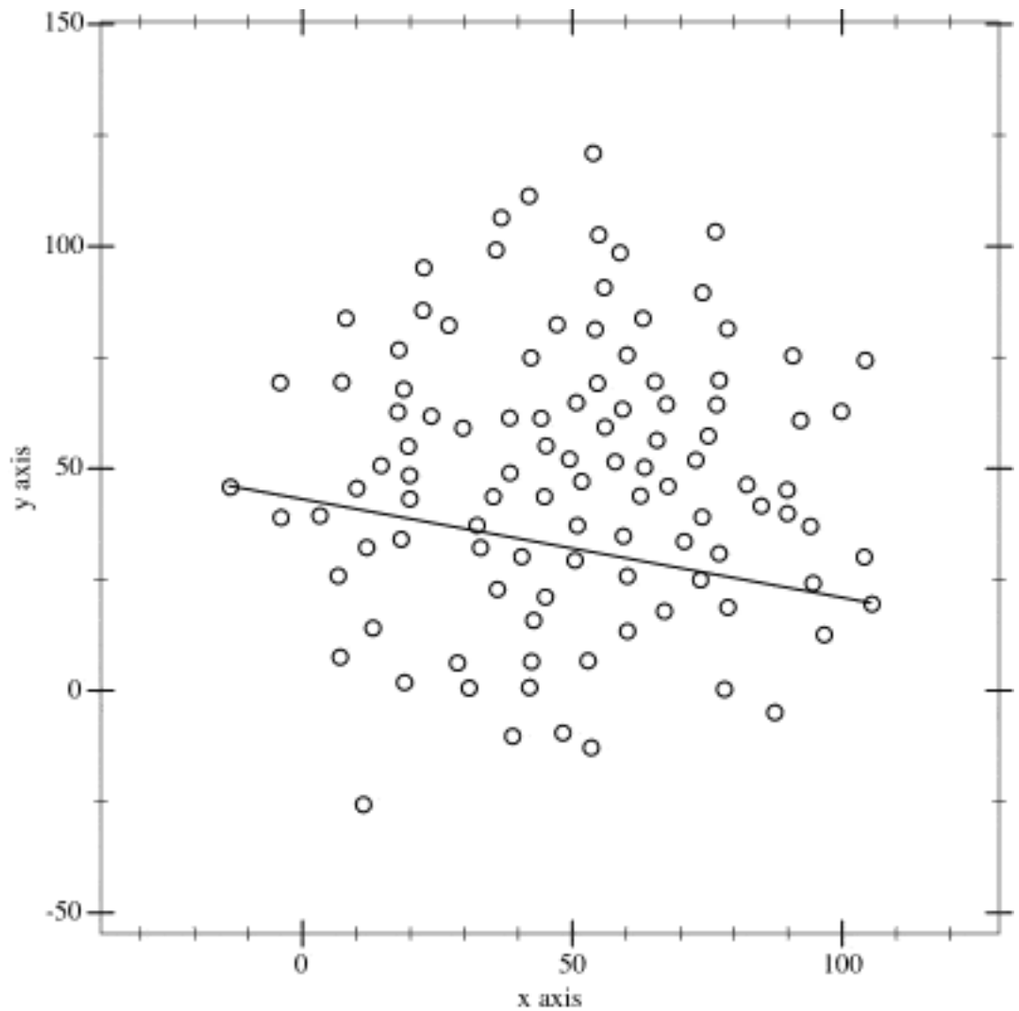


Runtime of Graham's can

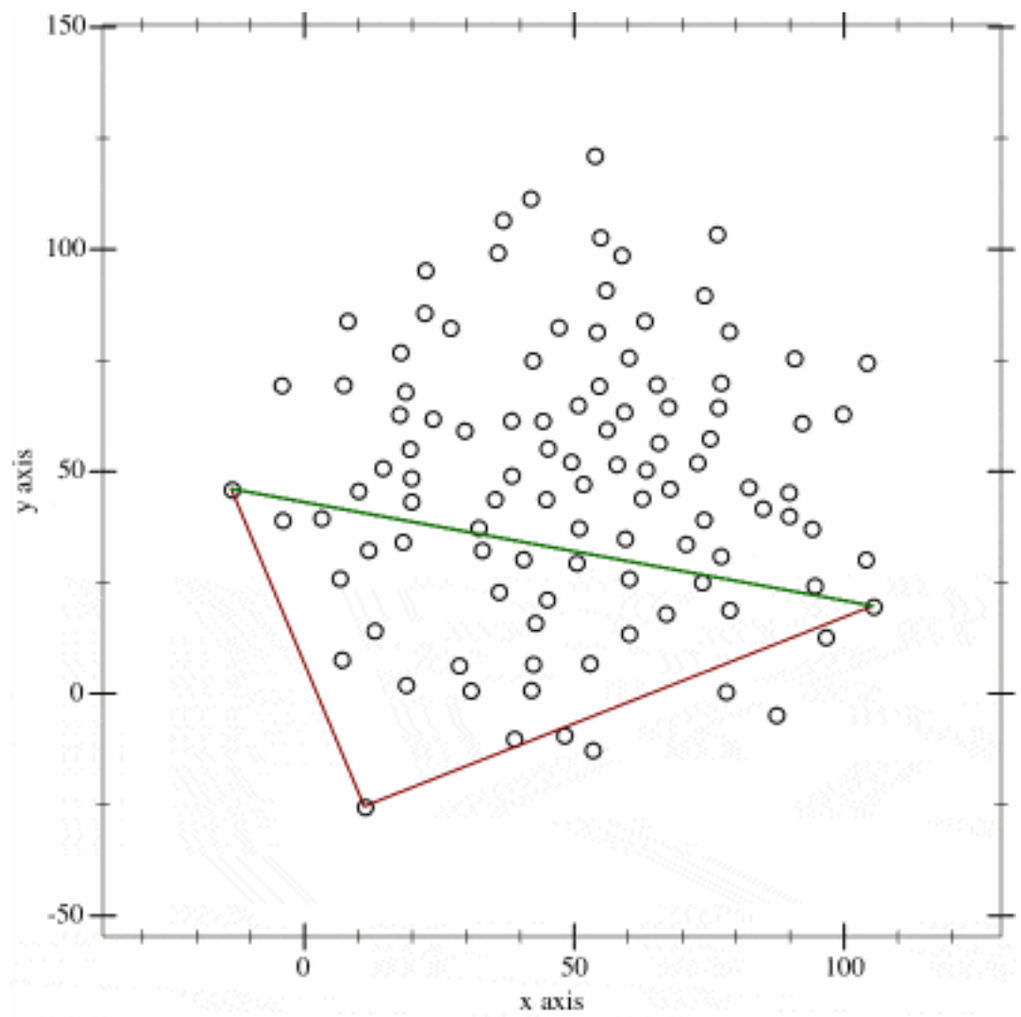
- Graham's scan is O(n log n) due to initial sort of angles.

Quick Hull Algorithm

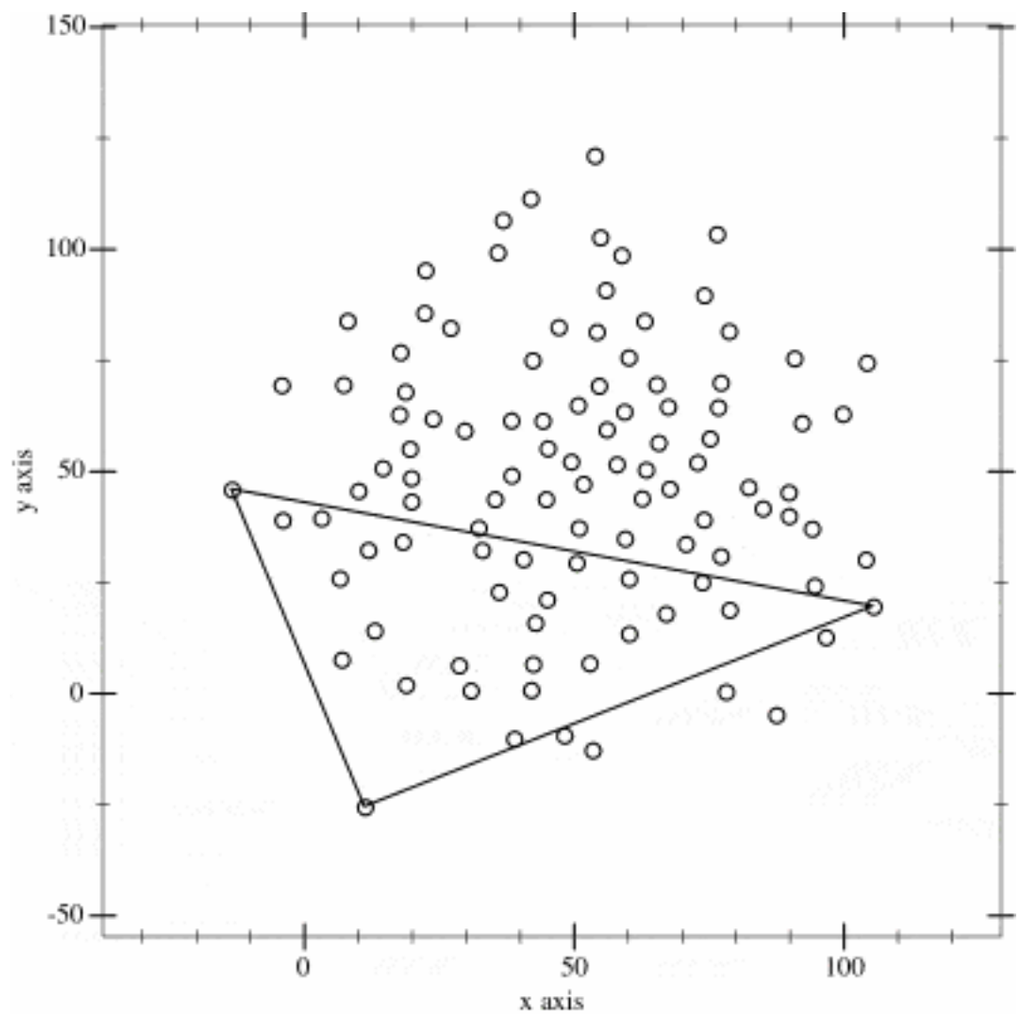
Quick hull example



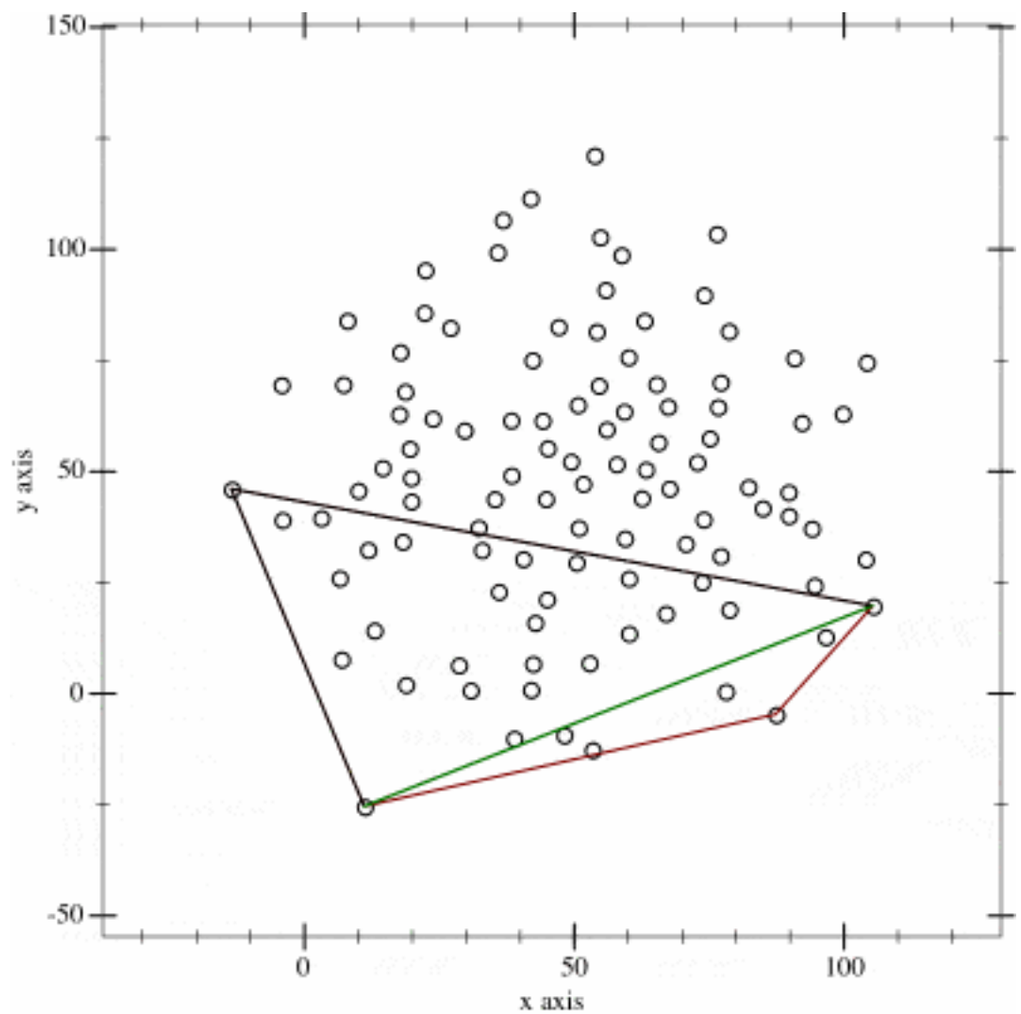
Quick hull example



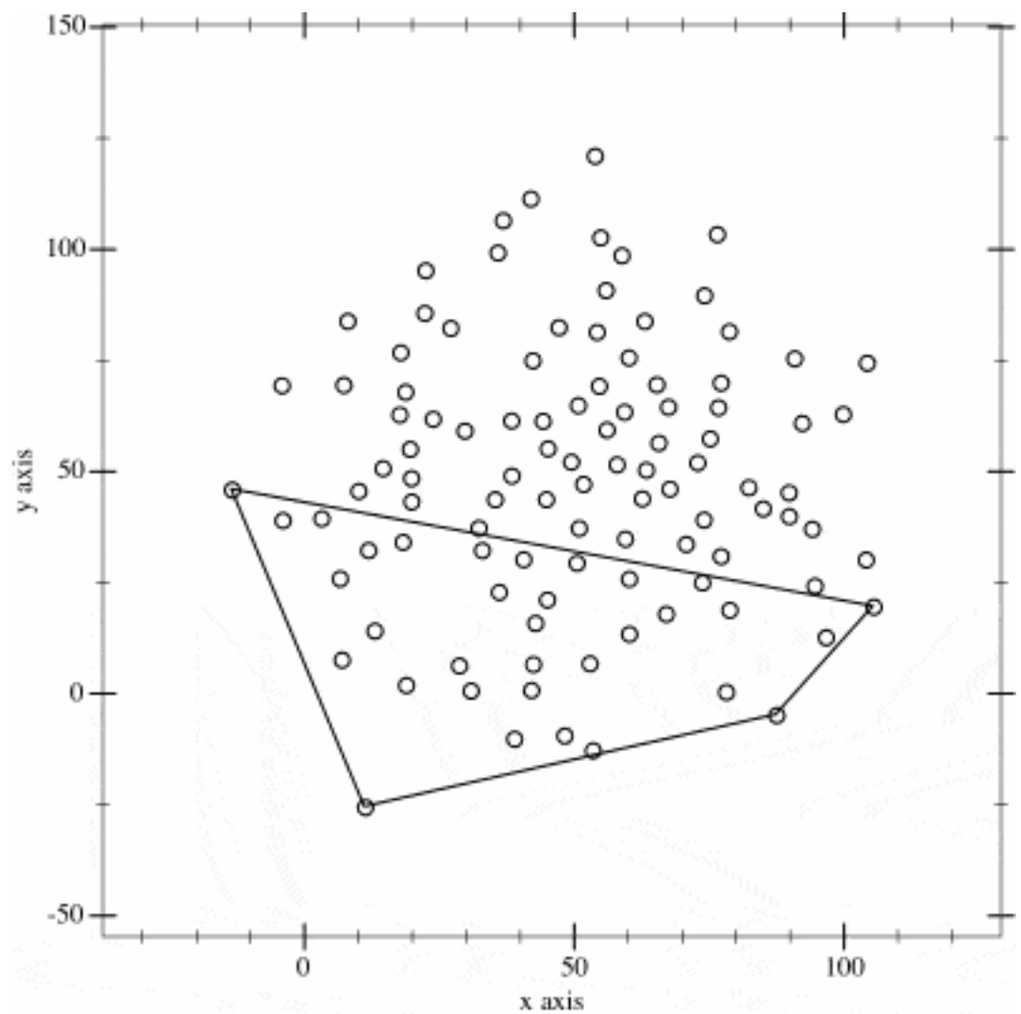
Quick hull example



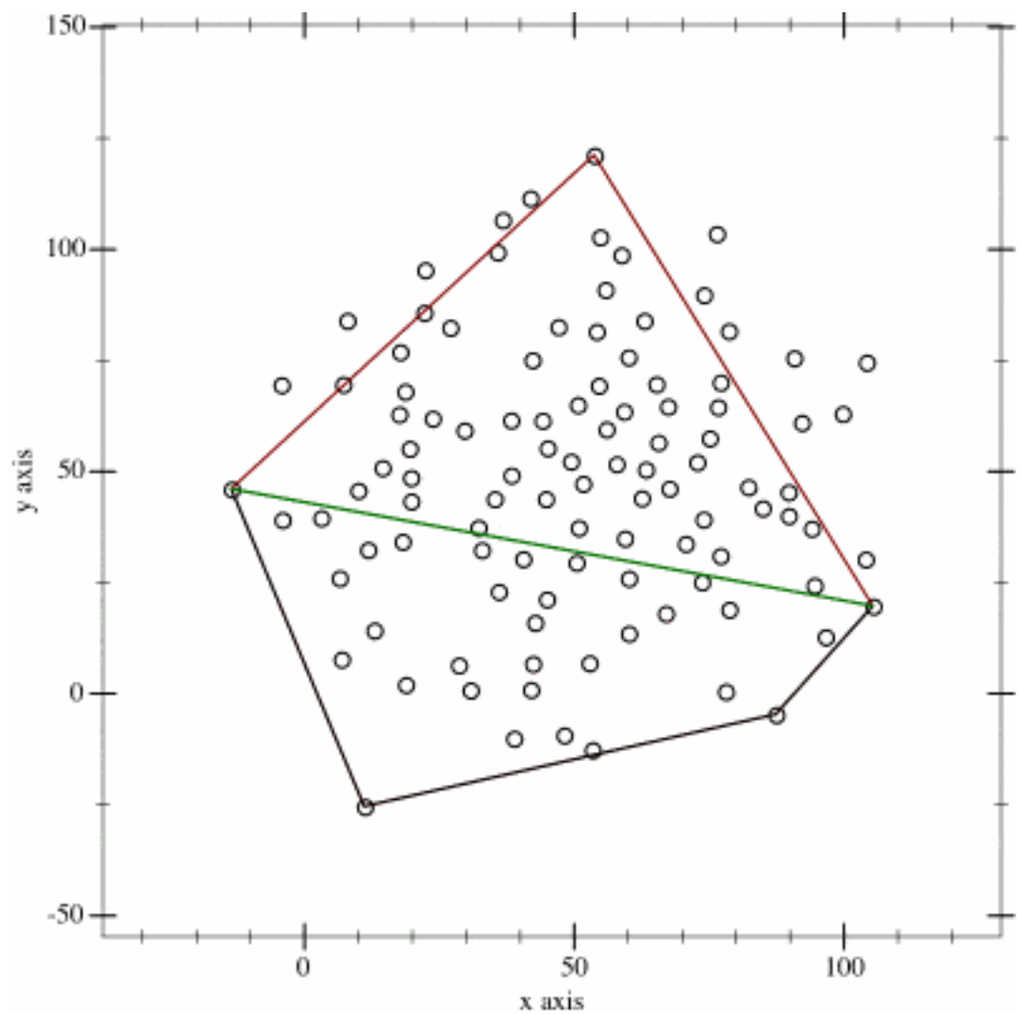
Quick hull example



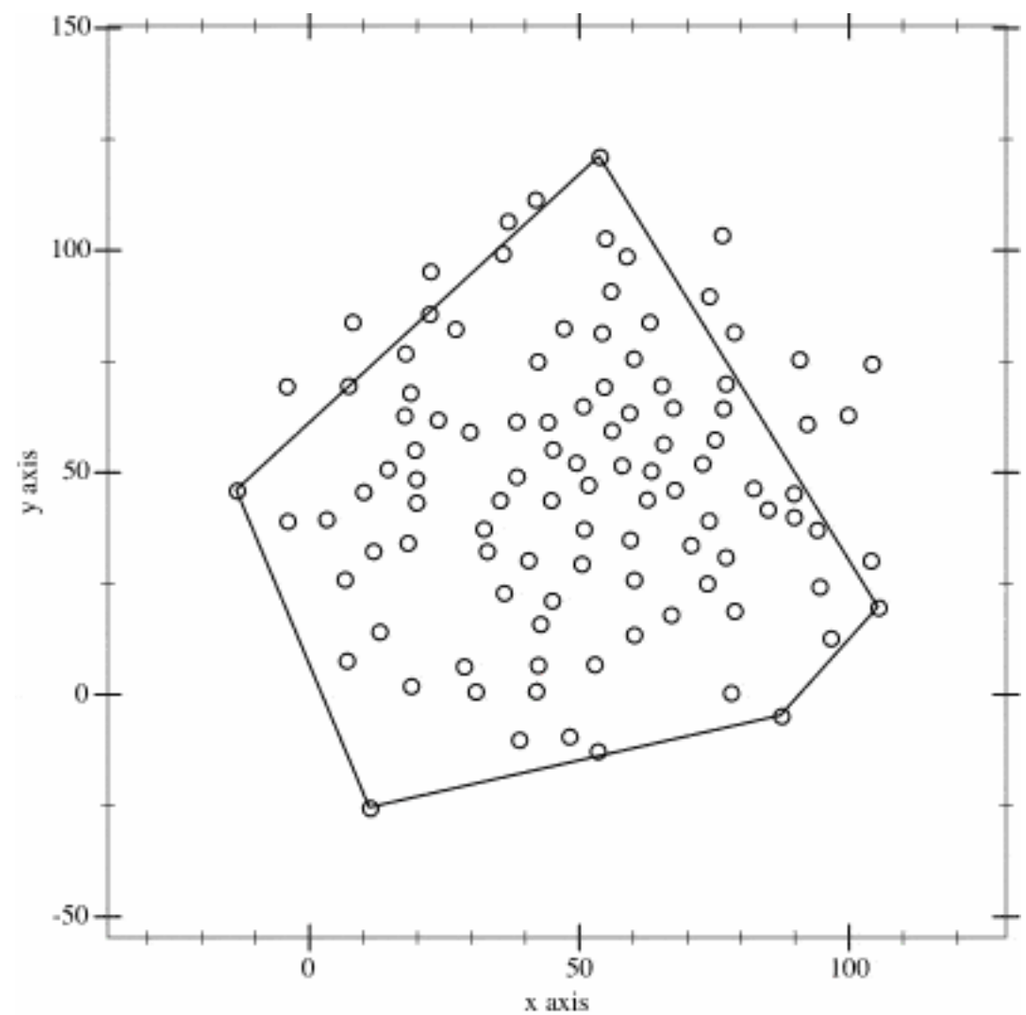
Quick hull example



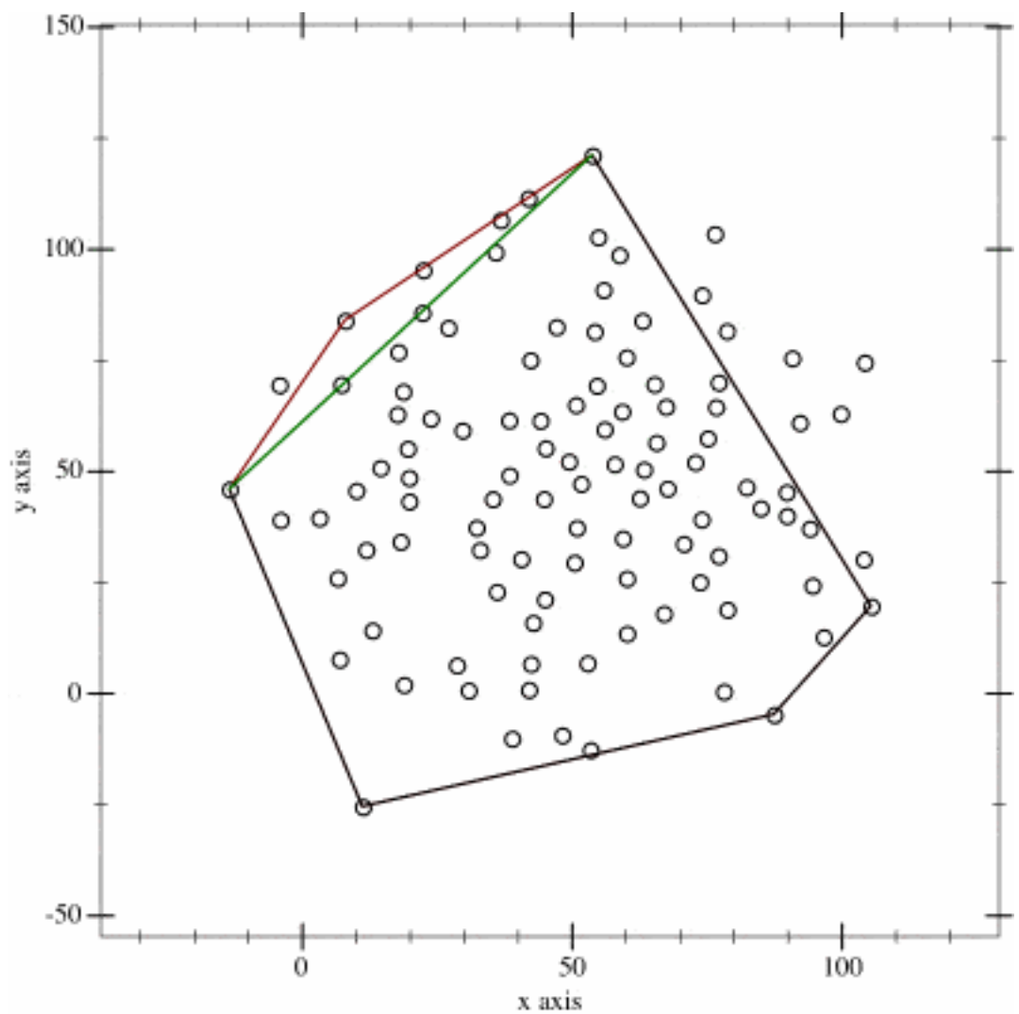
Quick hull example



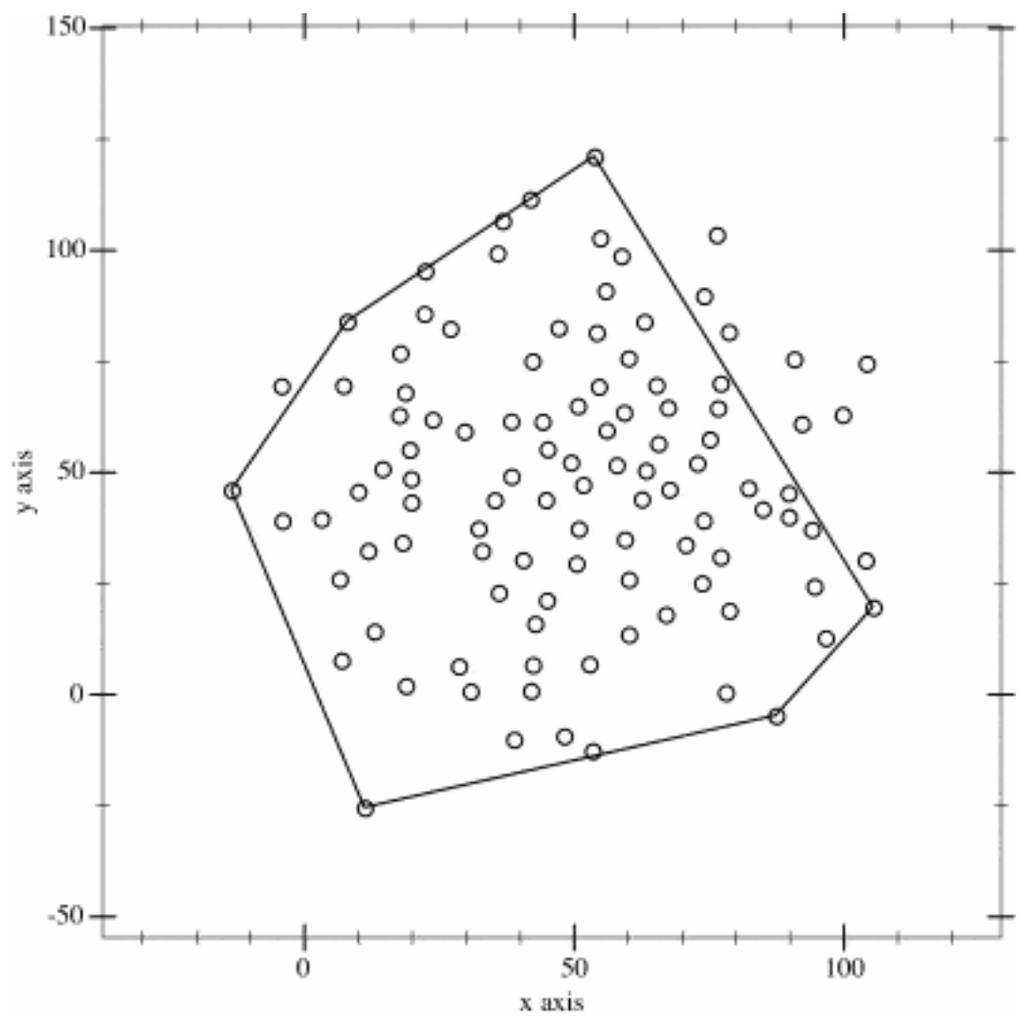
Quick hull example



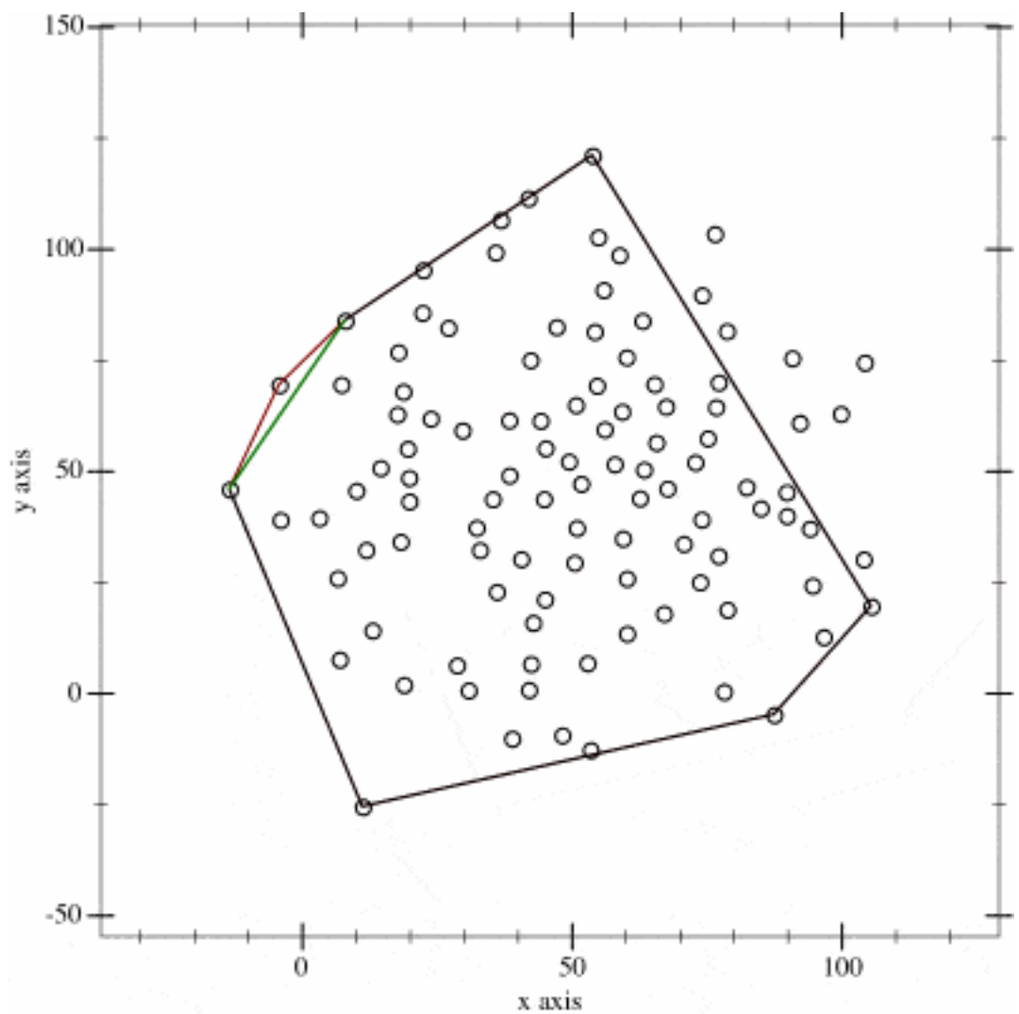
Quick hull example



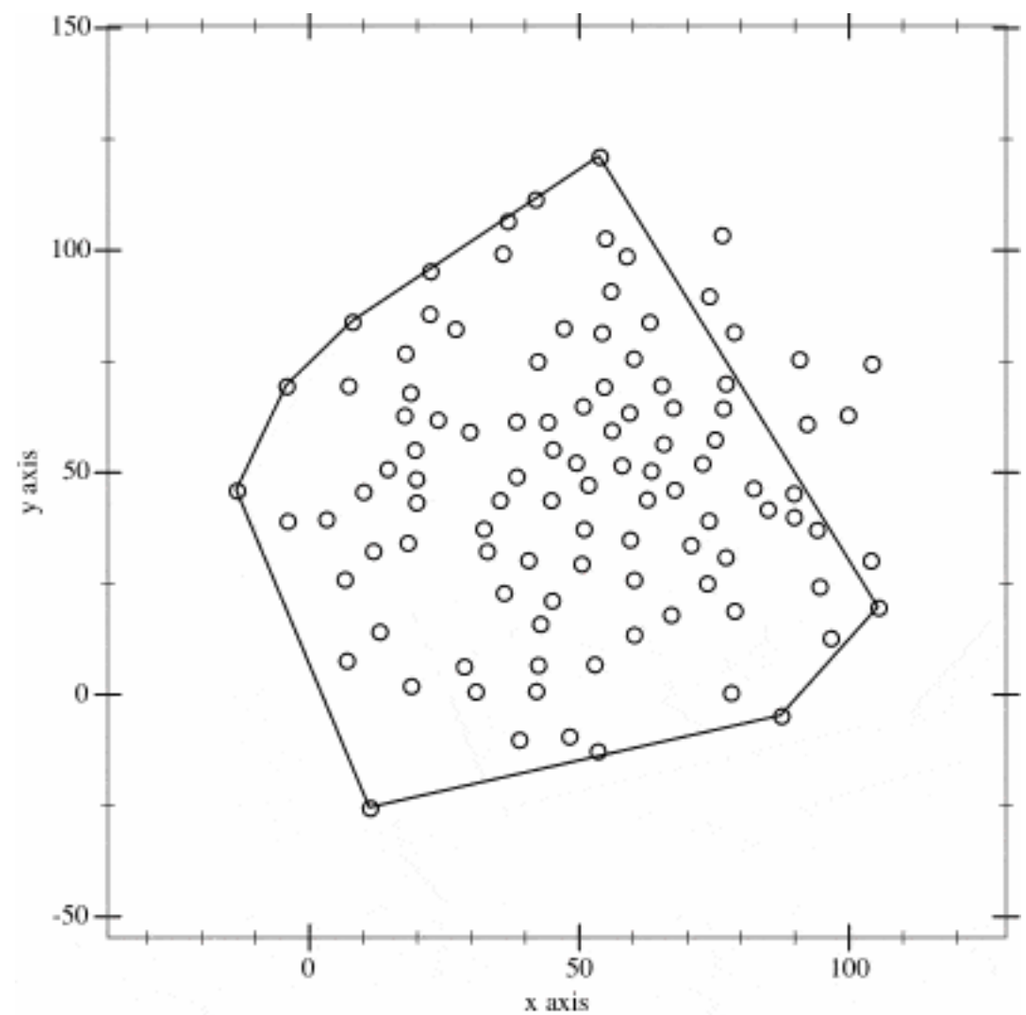
Quick hull example



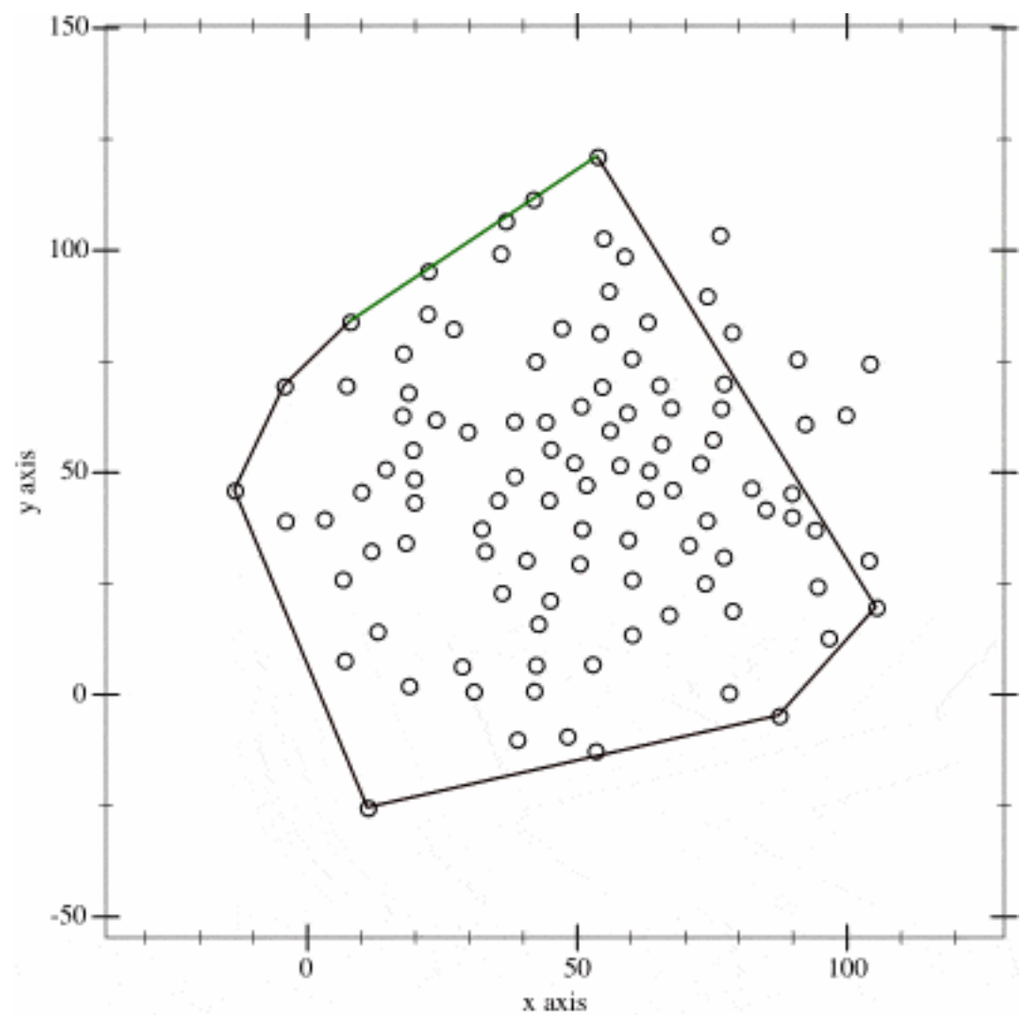
Quick hull example



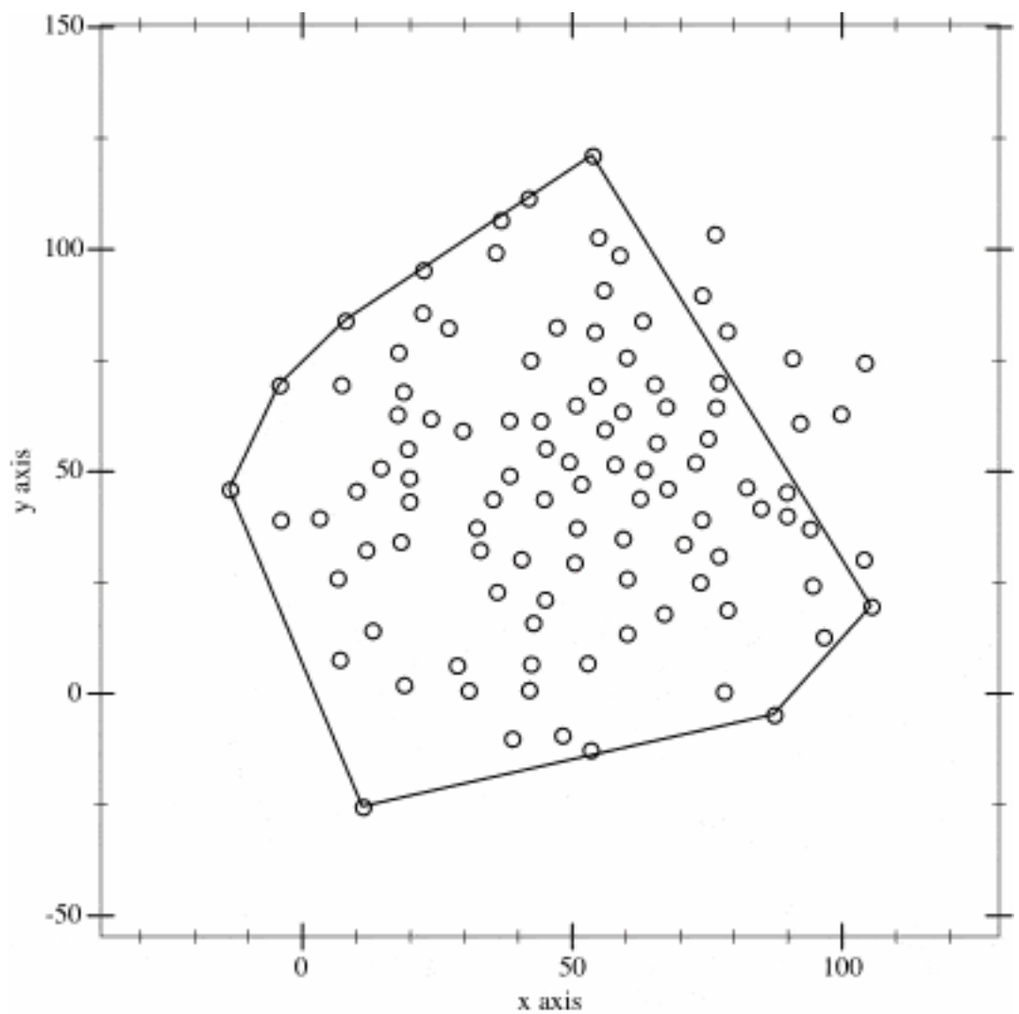
Quick hull example



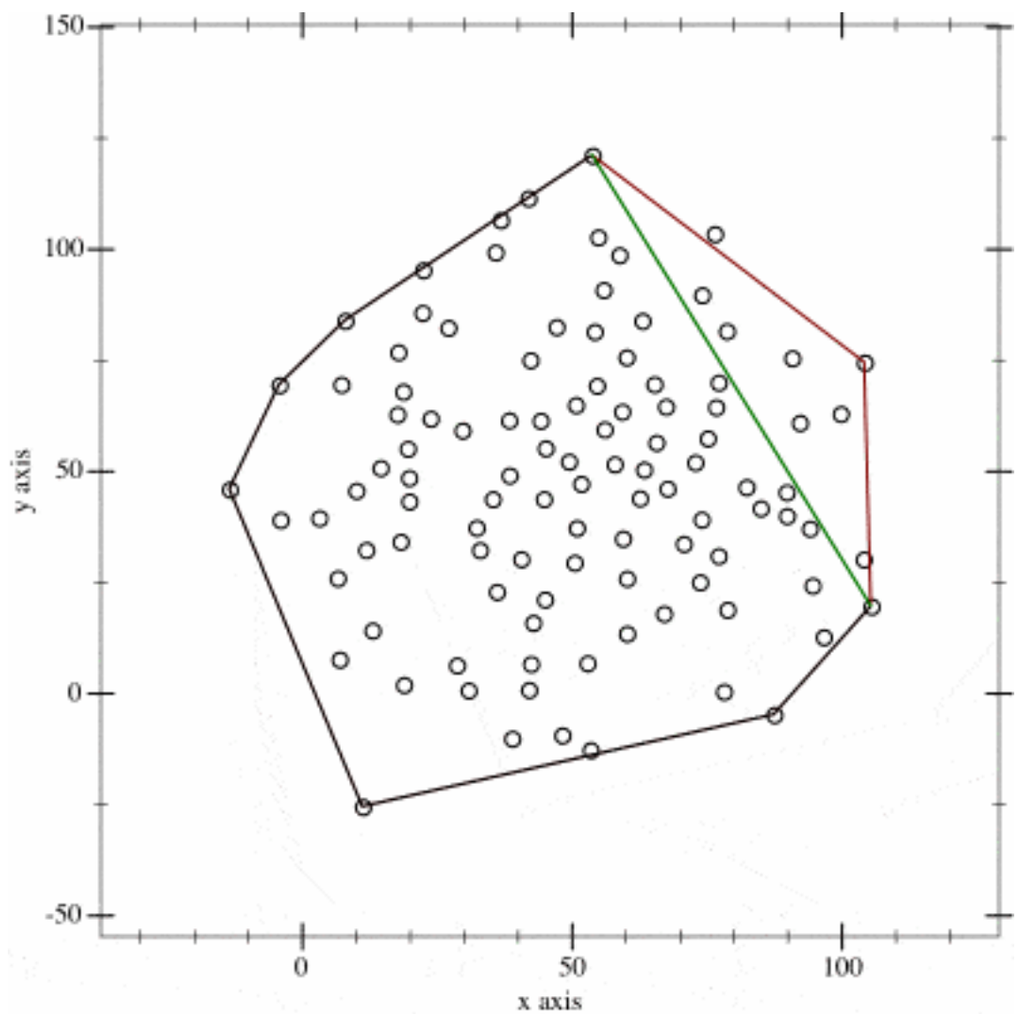
Quick hull example



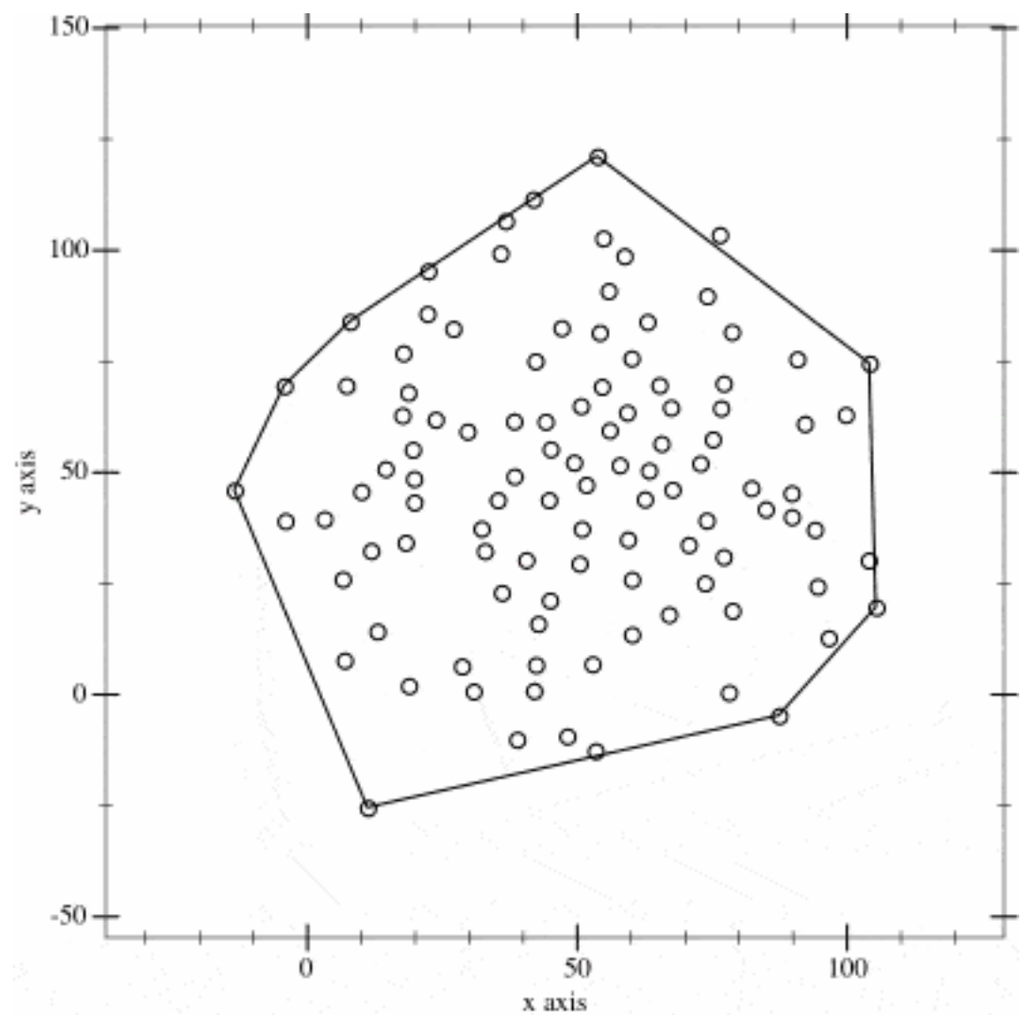
Quick hull example



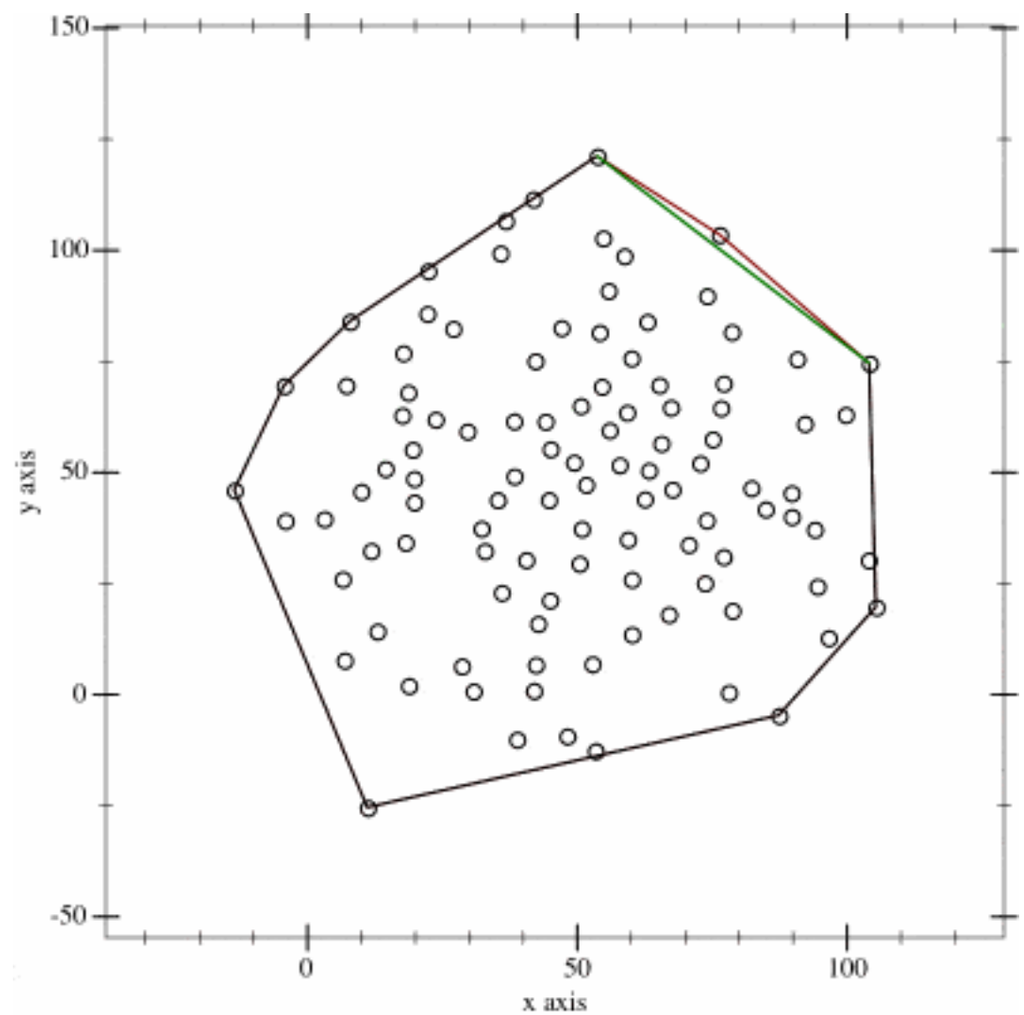
Quick hull example



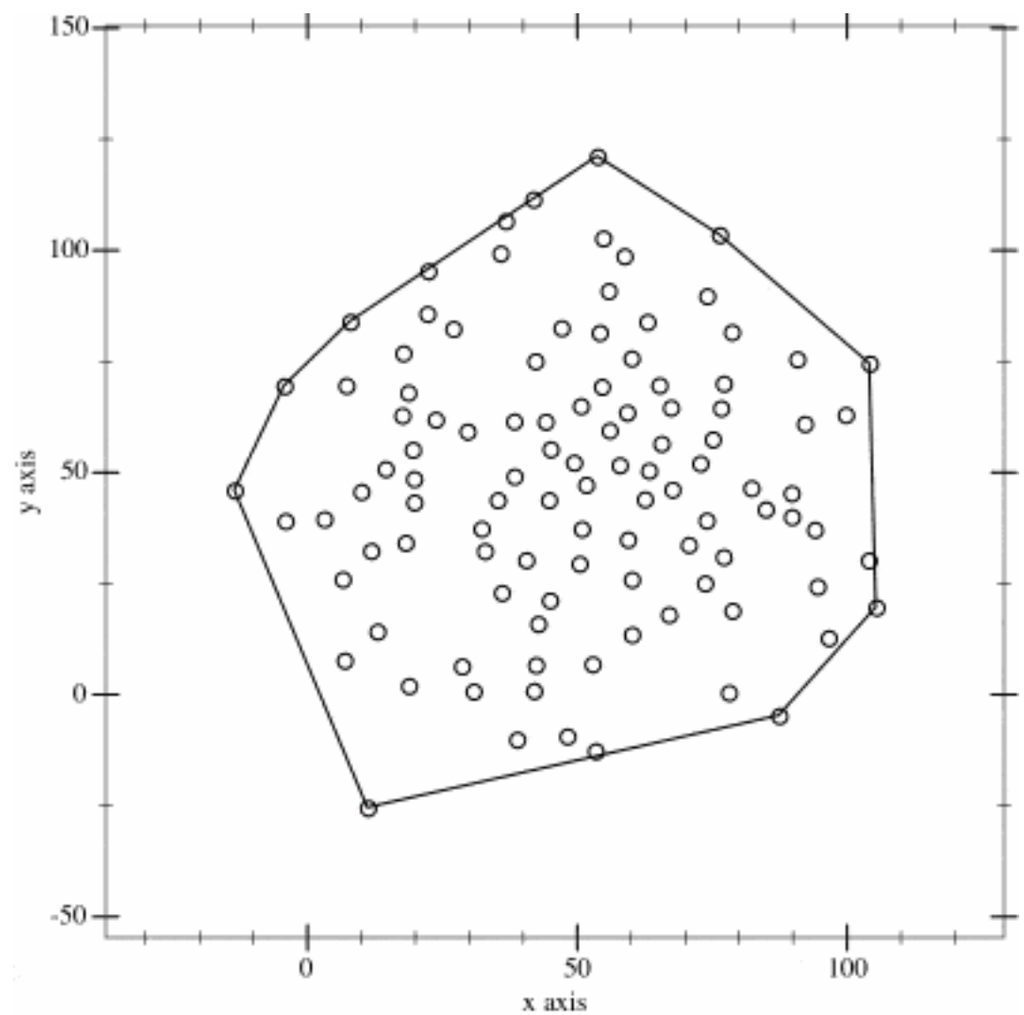
Quick hull example



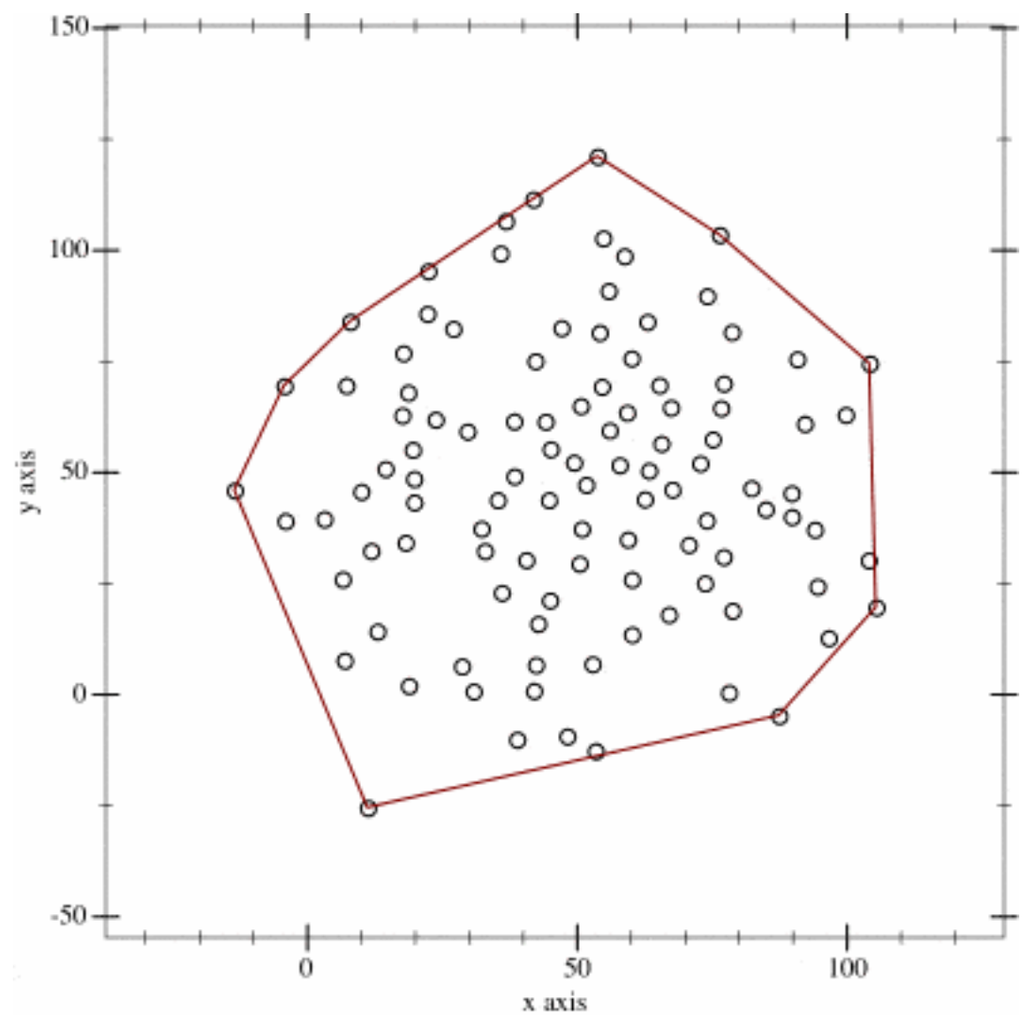
Quick hull example



Quick hull example



Quick hull example



QuickHull algorithm

```
Input = a set S of n points

Assume that there are at least 2 points in the input set S of points

QuickHull (S)
{
    // Find convex hull from the set S of n points
    Convex_Hull := {}

    Find left and right most points, say A & B
    Add A & B to convex hull
    Segment AB divides the remaining (n-2) points into 2 groups S1 and S2
         where S1 are points in S that are on the right side of the oriented line from A to B, and
               S2 are points in S that are on the right side of the oriented line         from B to A

    FindHull (S1, A, B)
    FindHull (S2, B, A)
}
```

QuickHull algorithm

```
FindHull (Sk, P, Q)
{
    // Find points on convex hull from the set Sk of points
    // that are on the right side of the oriented line from P to Q

    If Sk has no point, then return.

    From the given set of points in Sk, find farthest point, say C, from segment PQ

    Add point C to convex hull at the location between P and Q .

    Three points P, Q and C partition the remaining points of Sk into 3 subsets: S0, S1, and S2
           where S0 are points inside triangle PCQ,
                 S1 are points on the right side of the oriented line from  P to C
                 S2 are points on the right side of the oriented line from C to Q

    FindHull(S1, P, C)
    FindHull(S2, C, Q)

}

Output = Convex Hull
```

Smarter way to write

```
QuickHull (S, l, r)

     if S={ }    then return ()
else if S={l, r} then return (l, r)  // a single convex hull edge
else
    z = index of a point that is furthest (max distance) from xy.
    Let A be the set containing points strictly right of (x, z)
    Let B be the set containing points strictly right of (z, y)
    return {QuickHull (A, x, z) U (z) U QuickHull (B, z, y)}
```

Complexity

The partition is determined by the line passing through two distinct extreme points: the rightmost lowest `r` and the leftmost highest points `l`. Finding the extremes require `O(n)` time.

For the recursive function, it takes `n` steps to determine the extreme point `z`, but the cost of recursive calls depends on the sizes of set `A` and set `B`.

Best case. Consider the best possible case, when each partition is almost balanced. Then we have

`T(n) = 2 T(n/2) + O(n)`.

This is a familiar recurrence relation, whose solution is

`T(n) = O(n log(n))`.

This would occur with randomly distributed points.

Worst case. The worst case occurs when each partition is an extremely unbalanced. In that case the recurrence relation is

```
T(n) = T(n-1) + O(n)
     = T(n-1) + cn
```

Repeated expansion shows this is `O(n^2)`. Therefore, in the worst case the QuickHull is quadratic.

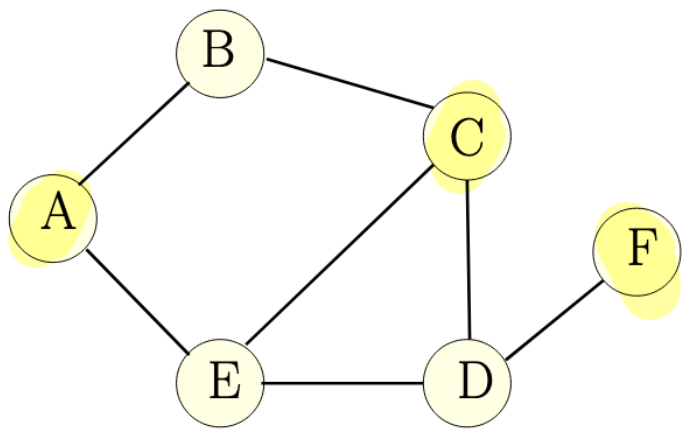
Maximum Weight Independent Set of a Tree
Dynamic Programming II

(ADA-2023)

Diptapriyo Majumdar

February 9, 2022

- Given a graph $G = (V, E)$, a set $S \subseteq V(G)$ is called independent set if for every $u, v \in S$, u and v are not adjacent in G.
- Tree is a connected graph with no cycle.



- Some independent sets are $\{D\}, \{B, E\}, \{A, C, F\}$.

- **Input:** A tree $T = (V, E)$ with $wt : V(T) \to \mathbb{R}^+$.
- **Goal:** Find a maximum weight independent set in T.
- For a subset $S \subseteq V(T)$, $weight(S) = \sum_{u \in S} wt(u)$.

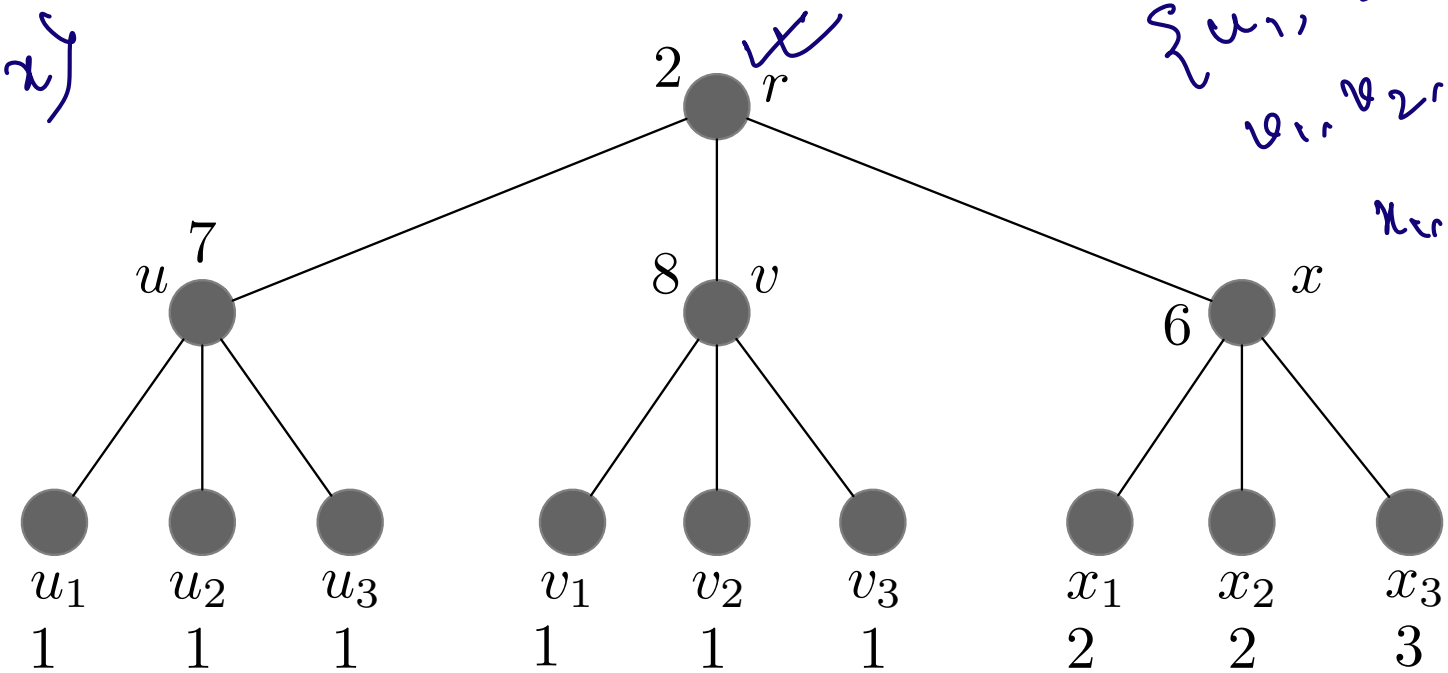
$\{u, v, x\}$

21

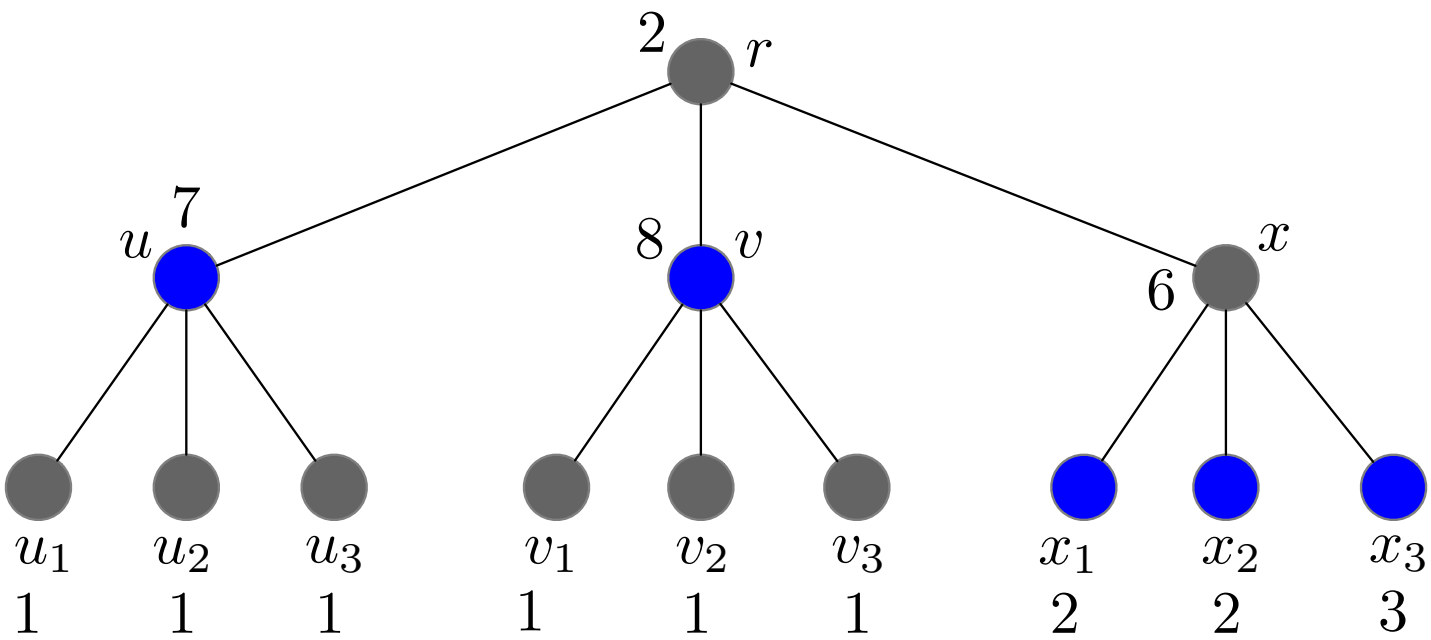
$\{u_1, u_2, u_3, v_1, v_2, v_3, x_1, x_2, x_3\}$

13

15



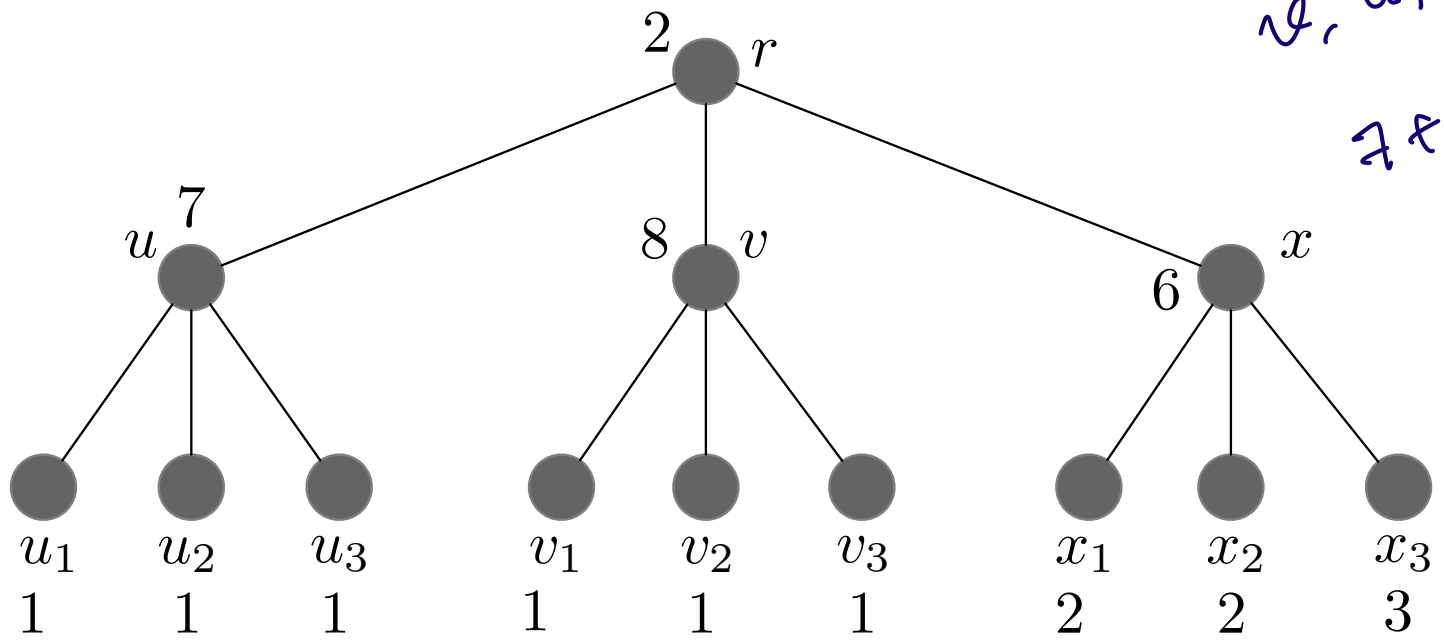
- **Input:** A tree $T = (V, E)$ with $wt : V(T) \to \mathbb{R}^+$.
- **Goal:** Find a maximum weight independent set in T.
- For a subset $S \subseteq V(T)$, $weight(S) = \sum_{u \in S} wt(u)$.



Greedy approach can give suboptimal solution

v, u, x

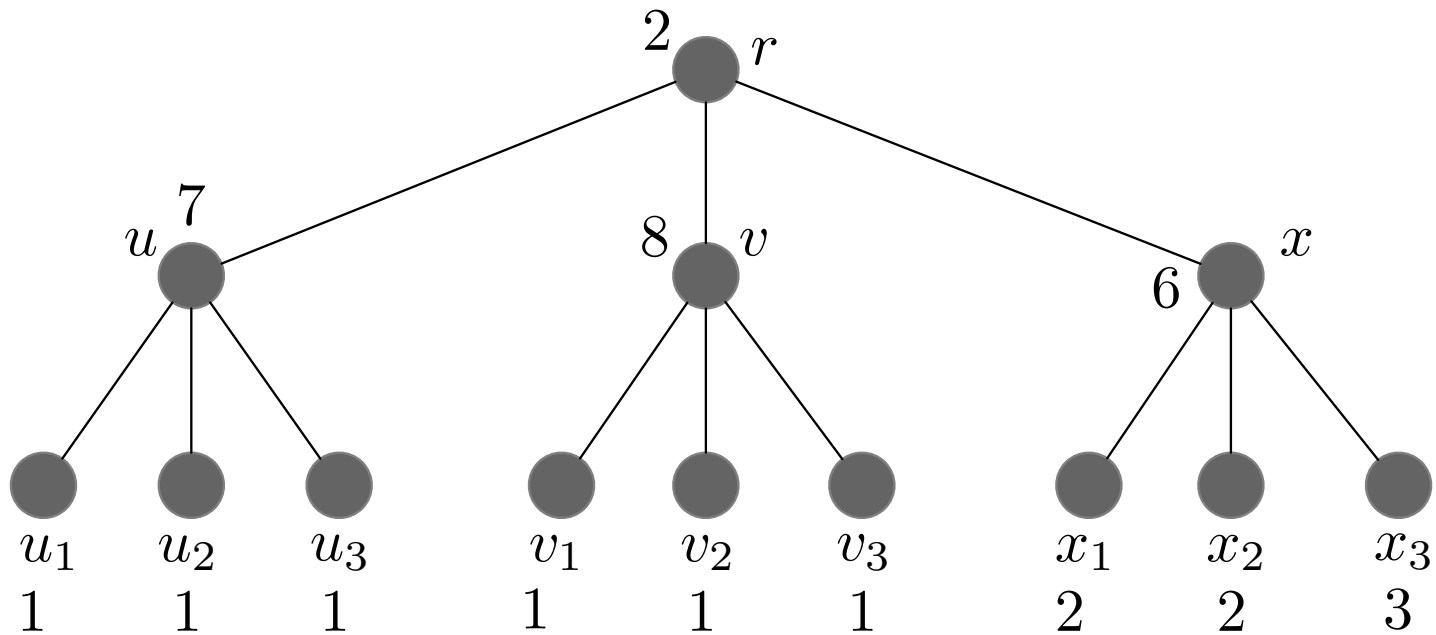
$7 + 8 + 6$



- Suppose the node u with maximum weight is picked first. v
- Then a node v with second maximum weight is picked so that u and v are nonadjacent.
- Repeat until no more nodes can be added.

Picks $\{u, v, x\}$

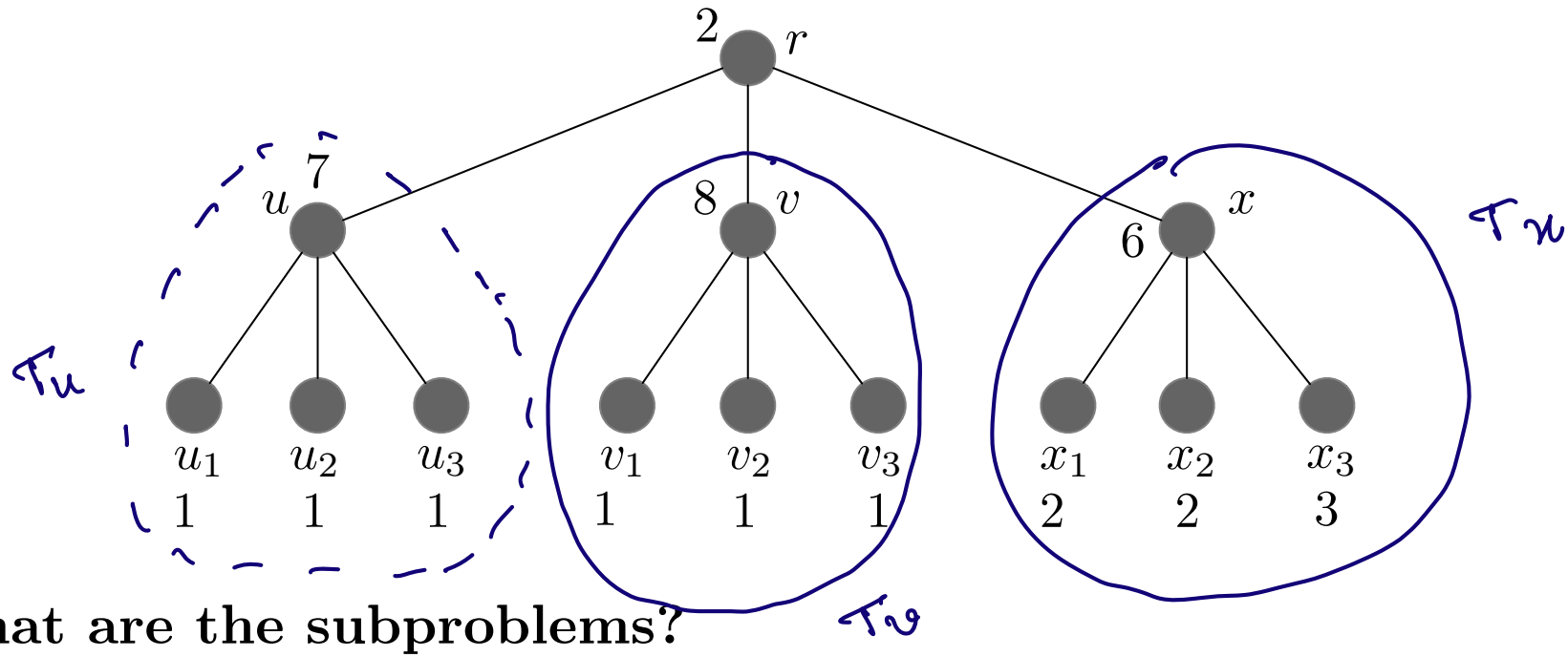
suboptimal.



- We can assume without loss of generality that a rooted tree is given to us.
- Postorder traversal gives an ordering of the vertices.
- The ordering first lists all the leaves, then their parents and so on. The root is the last vertex.

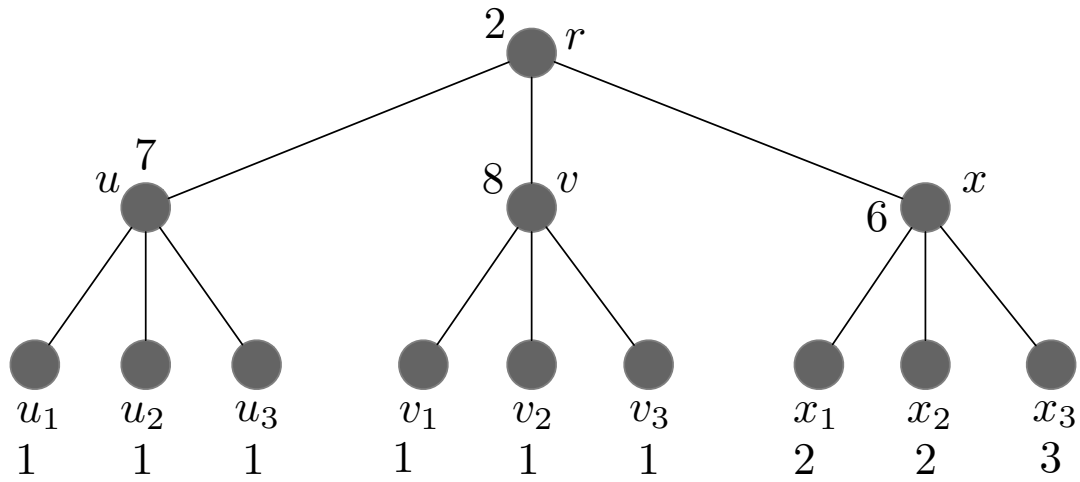
$\{u_1, u_2, u_3, u, v_1, v_2, v_3, v, x_1, x_2, x_3, x, r\}$

- We can assume without loss of generality that a rooted tree is given to us.
- Postorder traversal gives an ordering of the vertices.
- The ordering first lists all the leaves, then their parents and so on. The root is the last vertex.
- **There are two cases for any node.**
- **Case 1:** The node u is in a maximum weight independent set (i.e. in an optimal solution).
- **Case 2:** The node u is not in a maximum weight independent set (i.e. not in an optimal solution).



What are the subproblems?

- **Case 1:** The node u is in a maximum weight independent set (i.e. in an optimal solution).
- **Case 2:** The node u is not in a maximum weight independent set (i.e. not in an optimal solution).
- Let T_u be the (sub)tree rooted at u. Observe that $T = T_r$.

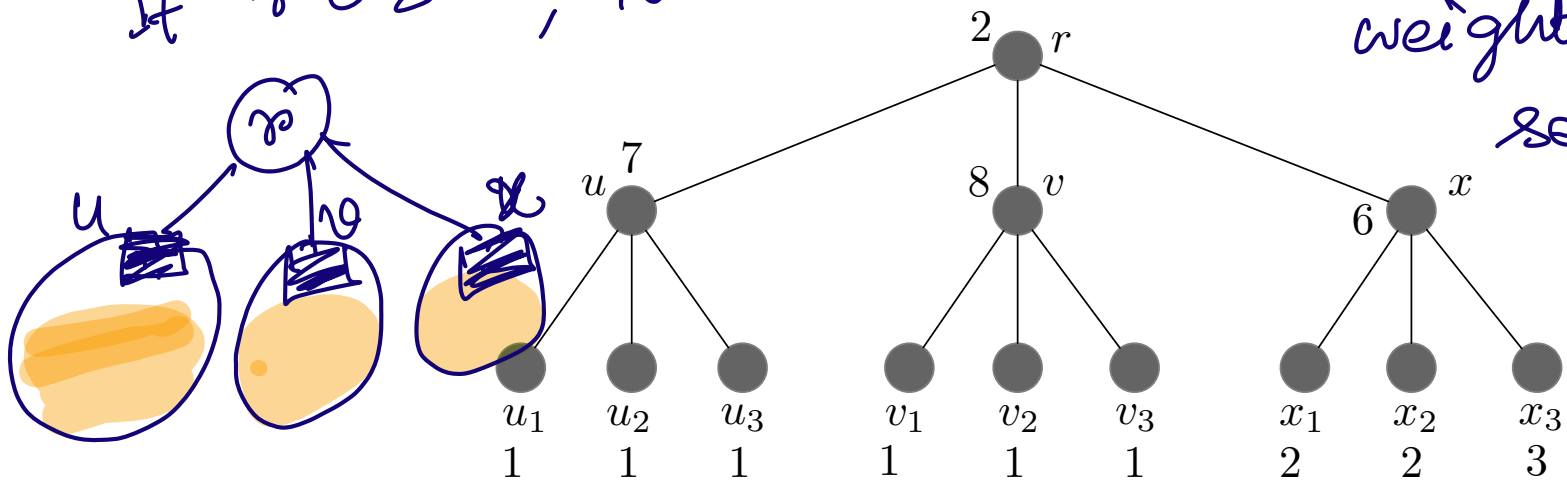


What are the subproblems?

- **Case 1:** $MIS(u, YES)$ is the total weight of a maximum weight independent set of T_u that contains u.
- **Case 2:** $MIS(u, NO)$ is the total weight of a maximum weight independent set of T_u not containing u.
- $MaxIS(u)$ is the total weight of a maximum weight independent set of T_u.

$MaxIS(u) = \max[MIS(u, YES), MIS(u, NO)]$

If $r \in S^*$, then S^* must contain maximum weight independent sets of T_u, T_v and T_x not containing u, v, w respectively.

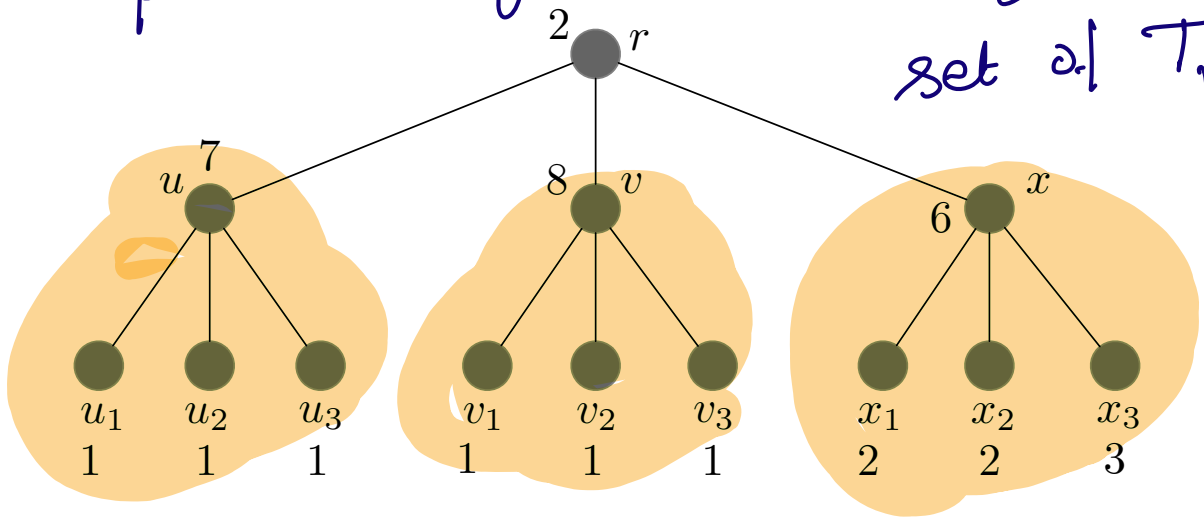


Optimal Substructure Property:

- **Case 1:** If an optimal independent set S^* of T_u contains u, then S^* must not contain any child v of u.
- Hence, for all children v of u, S^* must contain an optimal independent set of T_v not containing v.
- Why?. If for some child v, S^* contains v, then S^* is not independent.

Let for some child v of u, $S^* \supseteq P$ s.t. P is not a maximum weight independent set of T_v.

then replace P by Q_v → Q_v is an independent set of T_v and $v \notin Q_v$



Optimal Substructure Property:

- **Case 2:** If an optimal independent set S^* of T_u does not contain u, then S^* can contain some (if not all) child v of u.
- Hence, for all children v of u, S^* must contain an optimal independent set of T_v.
- Why?. Proof by contradiction

$$MIS(u, NO) = \sum_{v \in \text{children of } u} MaxIS(v)$$

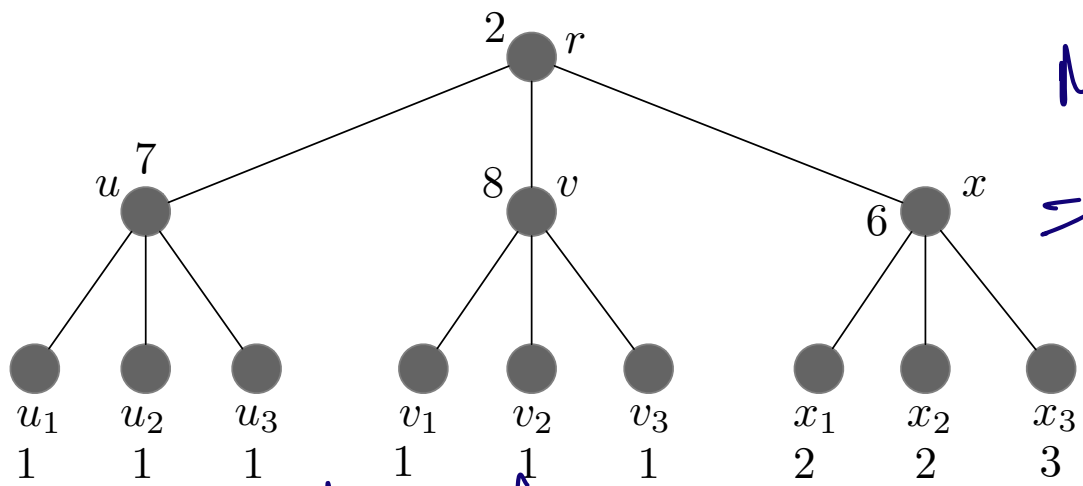
$MIS(r, NO)$

$= MaxIS(u)$

$+ MaxIS(v)$

$+ MaxIS(x)$

What is the recurrence relation? When we have the values $MIS(v, YES), MIS(v, NO)$ and $MaxIS(v)$ for all children v of u, how do we represent $MIS(r, YES)$ and $MIS(r, NO)$ using the values $MIS(v, YES), MIS(v, NO)$ and $MaxIS(v)$?

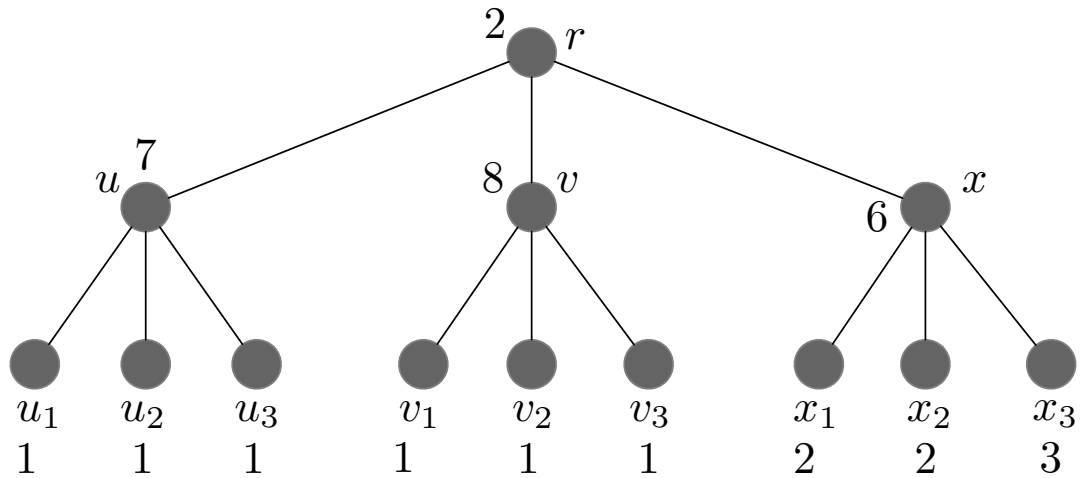


$MaxIS(u) = \max \begin{bmatrix} MIS(u, YES) \\ MIS(u, NO) \end{bmatrix}$

$MIS(u, YES)$ = value of maximum weight independent set of T_u containing u.

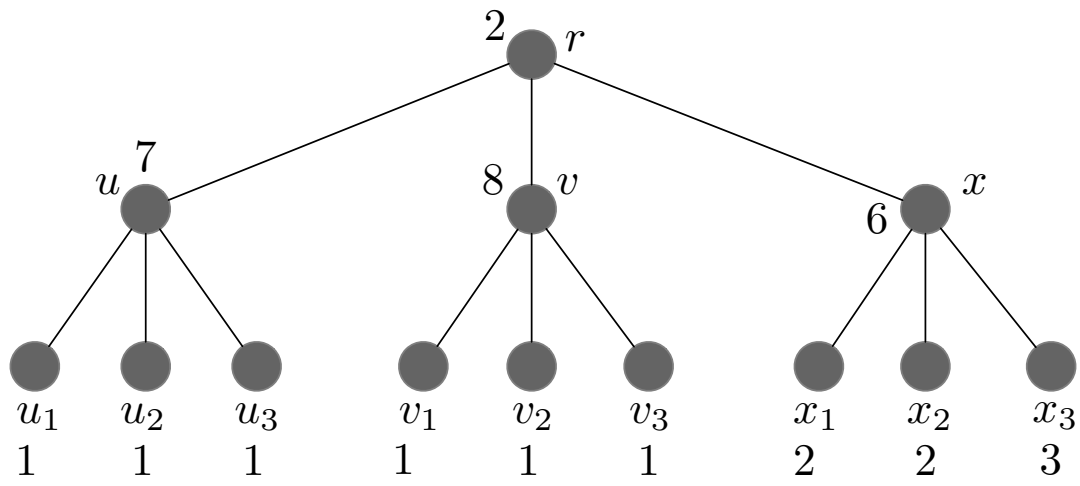
$MIS(u, NO)$ = value of maximum weight independent set of T_u not containing u.

What is the recurrence relation?



- $MIS(r, YES) = wt(r) + \sum_{v \text{ is a children of } r} MIS(v, NO)$.
- $MIS(r, NO) = \sum_{v \text{ is a children of } r} MaxIS(v)$.
- **Base Case:** If u is a leaf then $MIS(u, YES) = wt(u)$ and $MIS(u, NO) = 0$.
- Finally, $MaxIS(u) = \max\{MIS(u, YES), MIS(u, NO)\}$. any node
- Solution to the final problem is $MaxIS(r) = \max\{MIS(r, YES), MIS(r, NO)\}$.

Example



- The postorder traversal is $\{u_1, u_2, u_3, u, v_1, v_2, v_3, v, x_1, x_2, x_3, x, r\}$.
- Start filling up the table values $MIS(u, YES), MIS(u, NO)$ and $MaxIS(u)$ in the postorder traversal.
- How do we compute $MIS(u, YES)$ and $MIS(u, NO)$?

- If u is a leaf, then ~~what is~~ $MIS(u, YES) = wt(u)$ and $MIS(u, NO) = 0$.
- Then, for a leaf u, the $MaxIS(u) = \max\{MIS(u, YES), MIS(u, NO)\}$.
- When we compute $MIS(u, YES)$ and $MIS(u, NO)$, such that u is not a leaf, the values $MaxIS(v), MIS(v, NO), MIS(v, YES)$ are already computed for all children v of u.
- For any other node u,
  - $MIS(u, YES) = \cancel{wt(r)}\ wt(u) + \sum_{v \text{ is a children of } u} MIS(v, NO)$.
  - $MIS(u, NO) = \sum_{v \text{ is a children of } u} MaxIS(v)$.
  - $MaxIS(u) = \max(MIS(u, YES), MIS(u, NO))$.
- Output $MaxIS(r)$. r is the root.

- For every node u, what are the values do we store in the tables?

MIS(u, NO) MIS(u, YES) MaxIS(u).

- For every node u, how many other values do we need to access?

We need MIS(v, NO) for all children v of u

and MaxIS(v)

any child is a neighbor

Amortized analysis

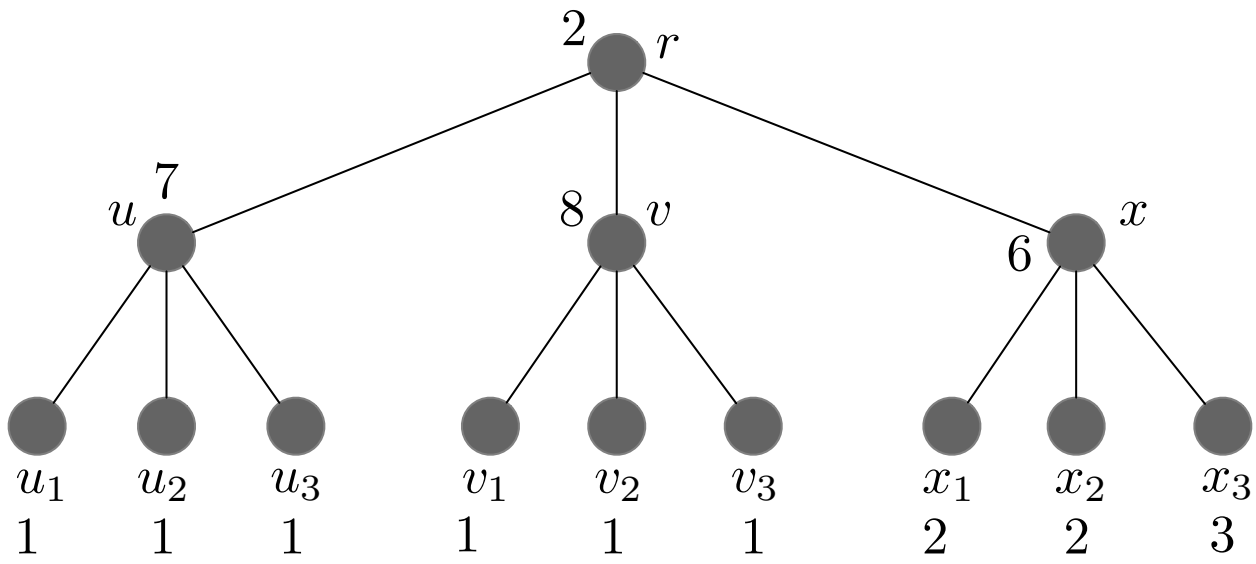
$$\sum_{u \in T} 3|N(u)| = \sum_{u \in T} 3 \deg_T(u) = 2|E(T)| * 3$$

$$\leq 2|V(T)| * 5$$

$\leq 6n$

- **Base Case:** If u is a leaf then $MIS(u, YES) = wt(u)$, $MIS(u, NO) = 0$ and $MaxIS(u) = \max\{MIS(u, YES), MIS(u, NO)\}$.
- **Recurrence Relation:** If u is not a leaf, then
  - $MIS(u, YES) = wt(r) + \sum_{v \text{ is a children of } u} MIS(v, NO)$.
  - $MIS(u, NO) = \sum_{v \text{ is a children of } u} MaxIS(v)$.
  - $MaxIS(u) = \max(MIS(u, YES), MIS(u, NO))$.
- **Solution to the final problem:** $MaxIS(r)$.

Finding explicit solution



- Start from r, whether $MaxIS(r) = MIS(r, YES)$ or $MaxIS(r) = MIS(r, NO)$.

References

- Jeff Erickson Lecture Notes Section 3.10.

Longest Increasing Subsequence
Dynamic Programming II

(ADA-2023)

Diptapriyo Majumdar

February 9, 2022

- **Input:** A sequence of n integers $X = \langle x_1, \ldots, x_n \rangle$.
- **Goal:** Find a subsequence Y of X such that Y is increasing and Y has maximum number of elements from X.
- **Example:**.

x_1	x_2	x_3	x_4	x_5	x_6
11	14	13	7	8	15

x_2	x_5	x_6
14	8	15
y_1	y_2	y_3

11 14 15

An increasing subsequence x_1, x_2, x_6

11	13	15
x_1	x_3	x_6

- **Input:** A sequence of n integers $X = \langle x_1, \ldots, x_n \rangle$.
- **Goal:** Find a subsequence Y of X such that Y is increasing and Y has maximum number of elements from X.

11 14 13 7 8 15

14 8 15

- **Input:** A sequence of n integers $X = \langle x_1, \ldots, x_n \rangle$.
- **Goal:** Find a subsequence Y of X such that Y is increasing and Y has maximum number of elements from X.
- It must be that $Y = \langle x_{\varphi(1)}, x_{\varphi(2)}, \ldots, x_{\varphi(k)} \rangle$ such that $x_{\varphi(i)} < x_{\varphi(i+1)}$.
- There are two cases.
- **Case 1:** x_n is not in an optimal subsequence.

Recursively solve for $\{x_1, x_2, \ldots, x_{n-1}\}$.

- **Case 2:** x_n is in an optimal subsequence.

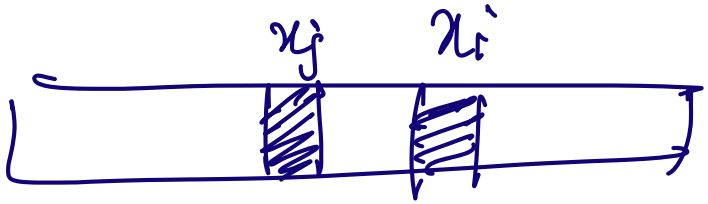
Longest increasing subsequence must contain x_n.

What are the subproblems?

$\langle x_1, x_2, \ldots, x_i \rangle$

- Define subproblems over $X_i = \langle \cancel{X_1, \ldots, X_i} \rangle$ and $X = X_n$.
- $LS(x_i)$ is the length of a longest subsequence of X_i that contains x_i.
- $LIS(x_i)$ is the length of a longest subsequence of X_i.
- **Case 2:** x_i is in an optimal subsequence of X_i.

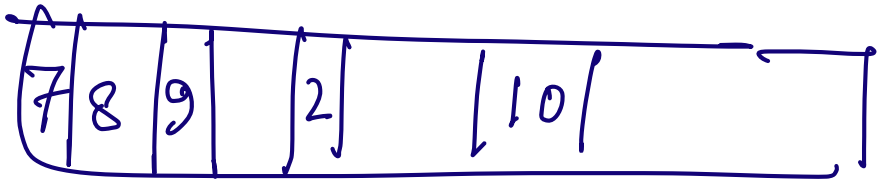
$$LS(x_i) = 1 + \max_{j<i: x_j < x_i} LS(x_j).$$



- Finally $LIS(x_n) = \max_{1 \le k \le n} LS(x_k)$.

$$LIS(x_i) = \max_{1 \le j \le i} LS(x_j)$$

Algorithm outline



- For every $1 \leq i \leq n$, initialize $Length(i) = 1$.
- Then, for every $i = 1, \ldots, n$ and for $j = i + 1, \ldots, n$.
  - If $x_j > x_i$, then
    $Length(j) = \max\{Length(j), Length(i) + 1\}$.
- Output $\max_{1 \leq i \leq n} Length(i)$.s

```
for (i=1, ..., n){
    for (j=i+1, i+2, ..., n)
        Length(j) = max [ length(j)
                          length(i)+1 ]
}
```

References

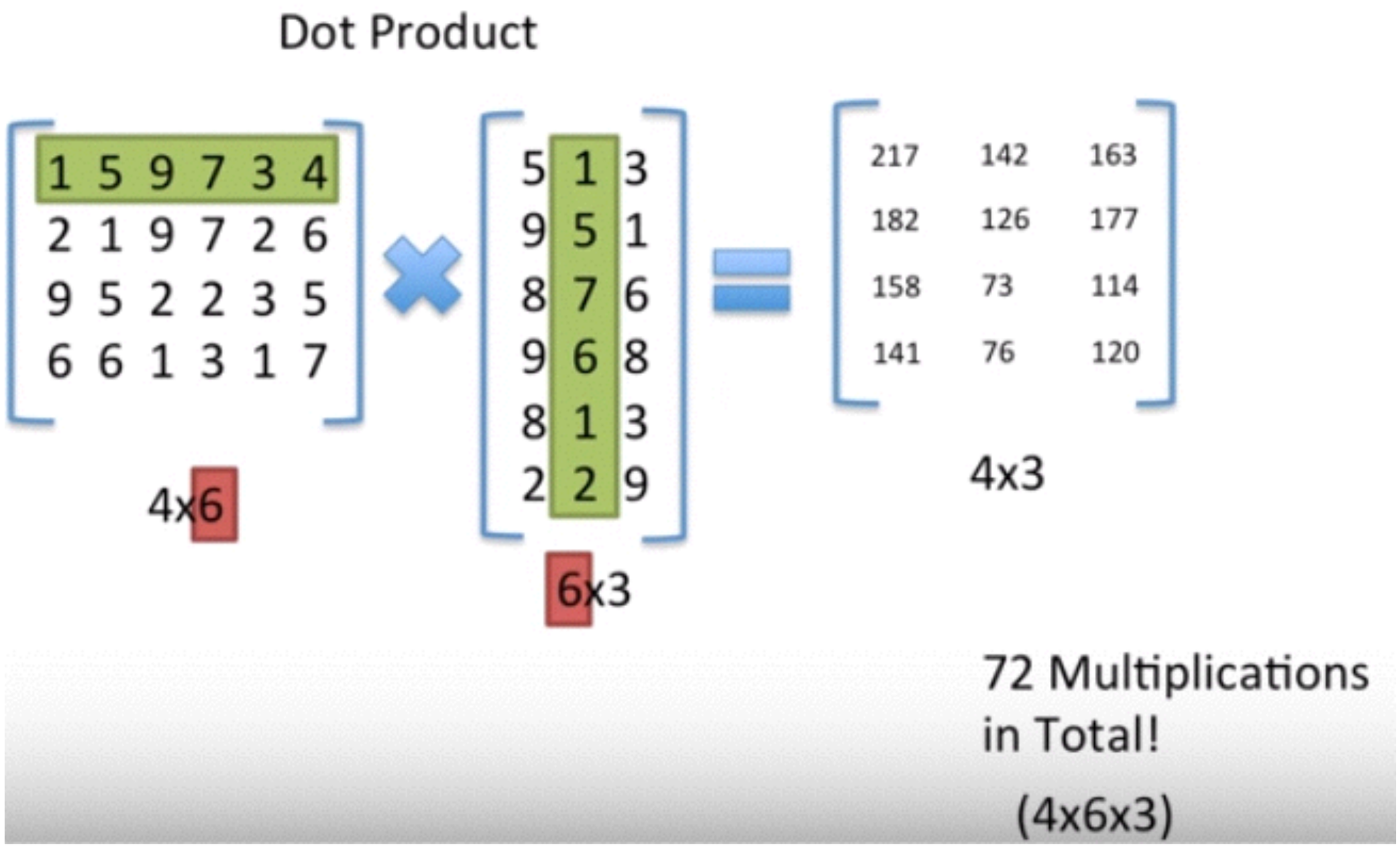
- Jeff Erickson Lecture Notes Section 3.6.
- Another lecture note is also attached.

Dynamic Programming - Part 2

Debarka Sengupta

Matrix Chain Multiplication

Revisit matrix multiplication



Matrix-chain Multiplication

- Suppose we have a sequence or chain A_1, A_2, …, A_n of *n* matrices to be multiplied
  - That is, we want to compute the product $A_1A_2\ldots A_n$

- There are many possible ways (parenthesizations) to compute the product

Matrix-chain Multiplication …contd

- Example: consider the chain A_1, A_2, A_3, A_4 of 4 matrices
  - Let us compute the product $A_1A_2A_3A_4$
- There are 5 possible ways:
  1. $(A_1(A_2(A_3A_4)))$
  2. $(A_1((A_2A_3)A_4))$
  3. $((A_1A_2)(A_3A_4))$
  4. $((A_1(A_2A_3))A_4)$
  5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication …contd

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

- Can you write the algorithm to multiply two matrices?

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p\times q$ and $q\times r$)
Result: Matrix $C_{p\times r}$ resulting from the product $A\cdot B$

MATRIX-MULTIPLY$(A_{p\times q}, B_{q\times r})$

```
1.  for i ← 1 to p
2.      for j ← 1 to r
3.          C[i, j] ← 0
4.          for k ← 1 to q
5.              C[i, j] ← C[i, j] + A[i, k] · B[k, j]
6.  return C
```

Scalar multiplication in line 5 dominates time to compute CNumber of scalar multiplications = *pqr*

Matrix-chain Multiplication …contd

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 =2{,}500$ scalar multiplications

    Total: 7,500
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 =50{,}000$ scalar multiplications

Total: 75,000

Matrix-chain Multiplication …contd

- Matrix-chain multiplication problem
  - Given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for i=1, 2, …, n, matrix A_i has dimension $p_{i-1} \times p_i$
  - Parenthesize the product $A_1 A_2 \ldots A_n$ such that the total number of scalar multiplications is minimized
- Brute force method of exhaustive search takes time exponential in n

Brute force

- For $n \geq 2$, a fully parenthesized matrix product is the product of 2 fully parenthesized matrix subproducts.
- The split can occur between k^{th} and $(k+1)^{th}$ matrices, for any $k = 1, 2, \ldots, n-1$
- So, the recurrence representing the total # of possible parenthesizations is:
  - $P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$
- Solution is tricky -- turns out, it grows as $\Omega\left(\frac{4^n}{n^{3/2}}\right)$
- Or, also true that it grows as $\Omega(2^n)$

Recursion (example of the first level)

- Consider the case multiplying these 4 matrices:
  - A: 2x4
  - B: 4x2
  - C: 2x3
  - D: 3x1

- 1. (A)(BCD) - This is a 2x4 multiplied by a 4x1,
  - so 2x4x1 = 8 multiplications, plus whatever work it will take to multiply (BCD).

- 2. (AB)(CD) - This is a 2x2 multiplied by a 2x1,
  - so 2x2x1 = 4 multiplications, plus whatever work it will take to multiply (AB) and (CD).

- 3. (ABC)(D) - This is a 2x3 multiplied by a 3x1,
  - so 2x3x1 = 6 multiplications, plus whatever work it will take to multiply (ABC).

Recursive formula

$$m(i,j) = \begin{cases} 0 & \text{If } i = j \\ \min_{i \le k < j}\{m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j\} & \text{If } i < j \end{cases}$$

Pseudocode

```
MATRIX-CHAIN(i, j)
    IF i = j THEN return 0
    m = ∞
    FOR k = i TO j − 1 DO
        q = MATRIX-CHAIN(i, k) + MATRIX-CHAIN(k + 1, j) + p_{i−1} · p_k · p_j
        IF q < m THEN m = q
    OD
    Return m
END MATRIX-CHAIN

Return MATRIX-CHAIN(1, n)
```

Recurrence relation

$$
\begin{aligned}
T(n) &= \sum_{k=1}^{n-1} (T(k) + T(n-k) + O(1)) \\
&= 2 \cdot \sum_{k=1}^{n-1} T(k) + O(n) \\
&\geq 2 \cdot T(n-1) \\
&\geq 2 \cdot 2 \cdot T(n-2) \\
&\geq 2 \cdot 2 \cdot 2 \ldots \\
&= 2^n
\end{aligned}
$$

Explaining he deduction applied to reach 2nd step

If n = 7, for k = 1, 2, .., 6 we get the following

1+ (7-1) = (1+6)
2+ (7-2) = (2+5)
3+ (7-3) = (3+4)
4+ (7-4) = (4+3)
5+ (7-5) = (5+2)
6+ (7-6) = (6+1)

On the right hand side it is simply (n-1) + (n-1) i.e., 2(n-1)

Recursion to DP through memorisation - trivial

```
MATRIX-CHAIN(i, j)
    IF T[i][j] < ∞ THEN return T[i][j]
    IF i = j THEN T[i][j] = 0, return 0
    m = ∞
    FOR k = i to j − 1 DO
        q = MATRIX-CHAIN(i, k) + MATRIX-CHAIN(k + 1, j) + p_{i−1} · p_k · p_j
        IF q < m THEN m = q
    OD
    T[i][j] = m
    return m
END MATRIX-CHAIN

return MATRIX-CHAIN(1, n)
```

Runtime complexity

- Quadratic due to the size of T i.e., the memo

Putting brackets optimally

$$A_1 \times A_2 \times A_3 \times A_4 \times A_5$$

4x10 10x3 3x12 12x20 20x7

Goal: Find the optimal way to multiply these matrices to perform the fewest multiplications.

Naïve Approach: Try them all, and pick the most optimal one.

Running time: $\Omega(4^n/n^{3/2})$ - 4^n dominates! Exponential

Substructure Opimality

There is a better way! Dynamic Programming!

Step 1: Check if the problem has Optimal Substructure

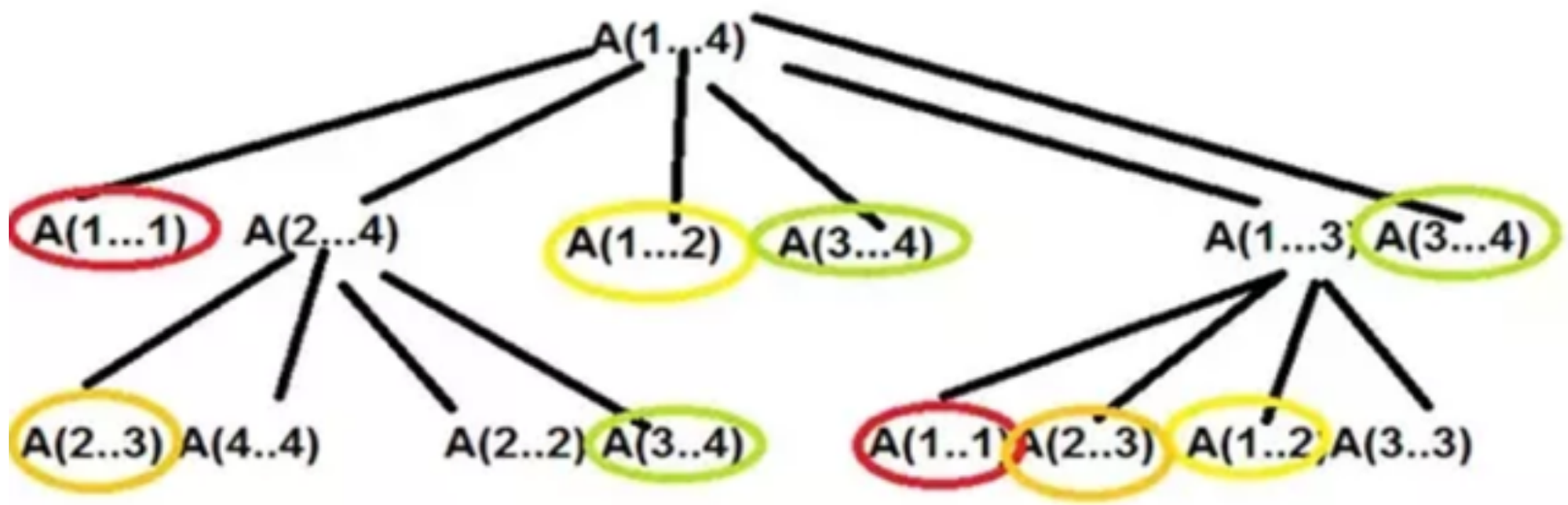
If we have an optimal solution for $A_{i...j}$

Assume the solution has the following parentheses:

$$(\underline{A_{i...k}})(\underline{A_{k+1...j}})$$

If there is a better way to multiply $(A_{i...k})$, then we would have a more optimal solution.

Overlapping subproblems



Recursive formula

Now we want to try out a bunch of values for 'k' in order to see what the best one is:

$$M[i,j] = \underset{100}{M[i,k]} + \underset{200}{M[k+1,j]} + \underset{2x3x4}{p_{i-1}p_kp_j}$$

Since we don't know what k is, we try this range of k:

$$\underset{2x3}{\overset{100}{(A_{i\ldots k})}}\underset{3x4}{\overset{200}{(A_{k+1\ldots j})}}$$

The minimum returned value is our solution!

$i \leq k < j$

Our Final Recursive Formula:

$$M[i,j] = \begin{cases} 0 \quad \text{if} \quad i=j \\ \min_{i \leq k < j} \{M[i,j] = M[i,k] + M[k+1,j] + p_{i-1}p_kp_j\} \end{cases}$$

An instance

Matrix Chain Multiplication

We want to start with i = j, then i<j starting with a spread of 1, working our way up

i \ j	1	2	3	4	5
1	0	120			
2	x	0	360		
3	x	x	0		
4	x	x	x	0	
5	x	x	x	x	0

$$M[i,j] = \begin{cases} 0 & \text{if } i=j \\ \min_{i \le k < j} \{M[i,j] = M[i,k] + M[k+1,j] + p_{i-1}p_kp_j\} \end{cases}$$

$$\underset{\substack{4\text{x}10 \\ p_0\ p_1}}{A_1} \times \underset{\substack{10\text{x}3 \\ p_1\ p_2}}{A_2} \times \underset{\substack{3\text{x}12 \\ p_2\ p_3}}{A_3} \times \underset{\substack{12\text{x}20 \\ p_3\ p_4}}{A_4} \times \underset{\substack{20\text{x}7 \\ p_4\ p_5}}{A_5}$$

$M[1,2] = \min_{1 \le k < 2} \{M[1,1] + M[1+1,2] + p_0p_1p_2\}$

$M[1,2] = \min_{1 \le k < 2} \{0 + 0 + 4\text{x}10\text{x}3\}$

$M[1,2] = 120$

$M[2,3] = \min_{2 \le k < 3} \{M[2,2] + M[2+1,3] + p_1p_2p_3\}$

$M[2,3] = \min_{2 \le k < 3} \{0 + 0 + 10\text{x}3\text{x}12\}$

$M[2,3] = 360$

$M[3,4] = \min_{3 \le k < 4} \{M[3,3] + M[3+1,4] + p_2p_3p_4\}$

$M[3,4] = \min_{3 \le k < 4} \{0 + 0 + 3\text{x}12\text{x}20\}$

An instance

$$M[i,j] = \begin{cases} 0 & \text{if } i=j \\ \min_{i \le k < j} \{M[i,j] = M[i,k] + M[k+1,j] + p_{i-1}p_kp_j\} \end{cases}$$

Matrix Chain Multiplication

We want to start with i = j, then i<j starting with a spread of 1, working our way up

i \ j	1	2	3	4	5
1	0	120			
2	x	0	360		
3	x	x	0	720	
4	x	x	x	0	1680
5	x	x	x	x	0

$$\underset{\substack{4\text{x}10 \\ p_0\ p_1}}{A_1} \times \underset{\substack{10\text{x}3 \\ p_1\ p_2}}{A_2} \times \underset{\substack{3\text{x}12 \\ p_2\ p_3}}{A_3} \times \underset{\substack{12\text{x}20 \\ p_3\ p_4}}{A_4} \times \underset{\substack{20\text{x}7 \\ p_4\ p_5}}{A_5}$$

$M[1,3] = \min_{1 \le k < 3}$

k=1

$= M[1,1] + M[1+1,3] + p_0p_1p_3$

$= 0 + 360 + 4\text{x}10\text{x}12$

$= 840$

k=2

$= M[1,2] + M[2+1,3] + p_0p_2p_3$

$= 120 + 0 + 4\text{x}3\text{x}12$

$= 264$

An instance

i \ j	1	2	3	4	5
1	0	120	264	1080	1344
2	x	0	360	1320	1350
3	x	x	0	720	1140
4	x	x	x	0	1680
5	x	x	x	x	0

Trace the solution

$$(A_1 \times A_2)((A_3 \times A_4)A_5)$$

A_1	A_2	A_3	A_4	A_5
4x10	10x3	3x12	12x20	20x7
$p_0\ p_1$	$p_1\ p_2$	$p_2\ p_3$	$p_3\ p_4$	$p_4\ p_5$

k=2

$M[1,5] = \underline{M[1,2]} + \underline{M[3,5]} + p_0p_2p_5$

k=4

$M[3,5] = M[3,4] + M[5,5] + p_2p_4p_5$

Edit Distance

Edit distance

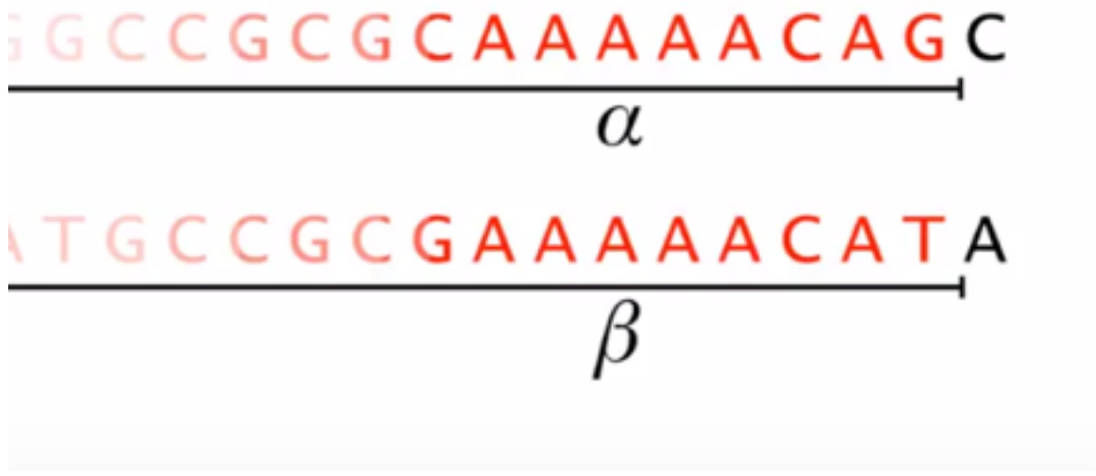
For X, Y where $|X| = |Y|$, *hamming distance* = minimum # substitutions needed to turn one into the other

For X, Y, *edit distance* = minimum # edits (substitutions, insertions, deletions) needed to turn one into the other

If $|X| = |Y|$ what can we say about the relationship between `editDistance`(X, Y) and `hammingDistance`(X, Y)?

`editDistance`(X, Y) $\leq$ `hammingDistance`(X, Y)

Substructure optimality



Recurrence

α C

β A

$$\texttt{edist}(\alpha\mathsf{C}, \beta\mathsf{A}) = \min \begin{cases} \texttt{edist}(\alpha, \beta) + 1 \\ \texttt{edist}(\alpha\mathsf{C}, \beta) + 1 \\ \texttt{edist}(\alpha, \beta\mathsf{A}) + 1 \end{cases}$$

Recurrence general

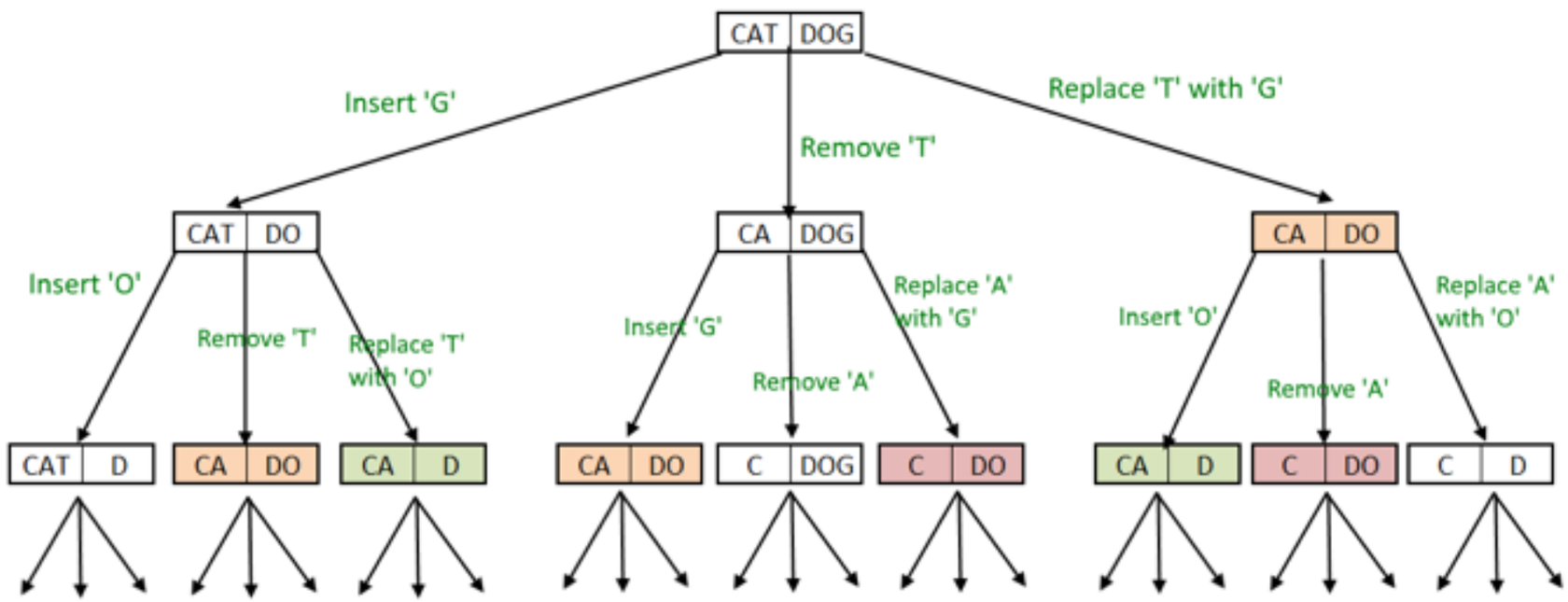
α x

β y

$$\texttt{edist}(\alpha x, \beta y) = \min \begin{cases} \texttt{edist}(\alpha, \beta) + \delta(x, y) \\ \texttt{edist}(\alpha x, \beta) + 1 \\ \texttt{edist}(\alpha, \beta y) + 1 \end{cases}$$

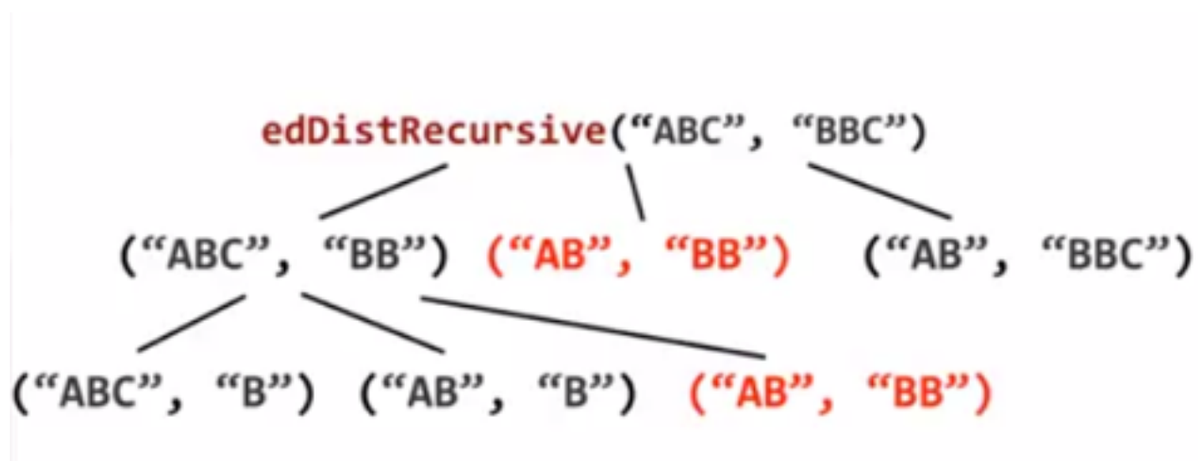
$\delta(x, y) = 0$ if $x = y$, or 1 otherwise

Recursion tree for an example



Recursive function

```
>>> import datetime as d
>>> st = d.datetime.now(); \
... edDistRecursive("Shakespeare", "shake spear"); \
... print (d.datetime.now()-st).total_seconds()
3
31.498284
```

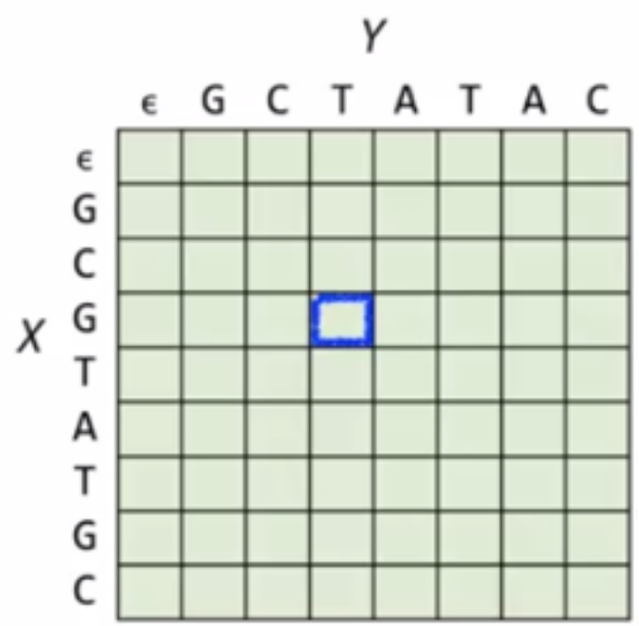


Repetitions

```
n = 0
def edDistRecursive(a, b):
    global n
    if len(a) == 0:
        return len(a)
    if len(b) == 0:
        return len(b)
    if a == 'Shake' and b == 'shake':
        n += 1
    delt = 1 if a[-1] != b[-1] else 0
    return min(edDistRecursive(a[:-1], b[:-1]) + delt,
               edDistRecursive(a[:-1], b) + 1,
               edDistRecursive(a, b[:-1]) + 1)
```

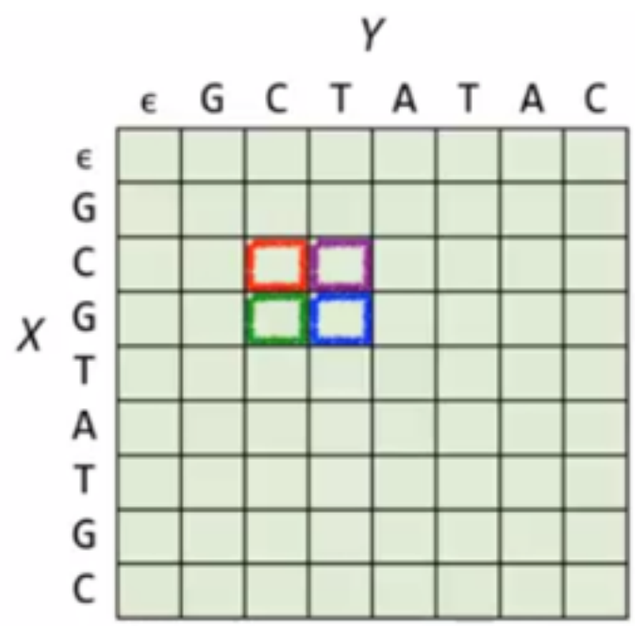
```
>>> edDistRecursive("Shakespeare", "shake spear")
3
>>> n
8989
```

Instance



$$\boxed{\texttt{edist}(\alpha x, \beta y)} = \min \begin{cases} \texttt{edist}(\alpha, \beta) + \delta(x, y) \\ \texttt{edist}(\alpha x, \beta) + 1 \\ \texttt{edist}(\alpha, \beta y) + 1 \end{cases}$$

Instance



$$\boxed{\texttt{edist}(\alpha x, \beta y)} = \min \begin{cases} \boxed{\texttt{edist}(\alpha, \beta) + \delta(x, y)} \\ \boxed{\texttt{edist}(\alpha x, \beta) + 1} \\ \boxed{\texttt{edist}(\alpha, \beta y) + 1} \end{cases}$$

Instance

		Y						
		ϵ	G	C	T	A	T	A	C
	ϵ	0	1	2	3				
	G	1	0	1	2				
	C	2	1	0	1				
X	G	3	1	1	1				
	T								
	A								
	T								
	G								
	C								

$$\boxed{\texttt{edist}(\alpha x, \beta y)} = \min \begin{cases} \boxed{\texttt{edist}(\alpha, \beta) + \delta(x, y)} & = 0 + 1 = 1 \\ \boxed{\texttt{edist}(\alpha x, \beta) + 1} & = 1 + 1 = 2 \\ \boxed{\texttt{edist}(\alpha, \beta y) + 1} & = 1 + 1 = 2 \end{cases}$$

End

Application of BFS – Testing Bipartiteness
Graph Algorithms

Diptapriyo Majumdar

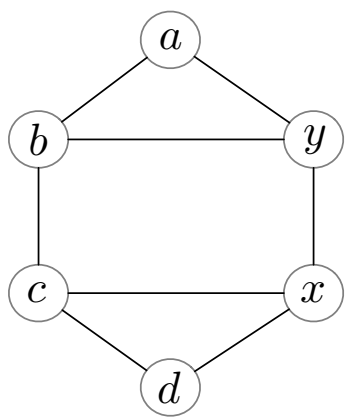
March 16, 2023

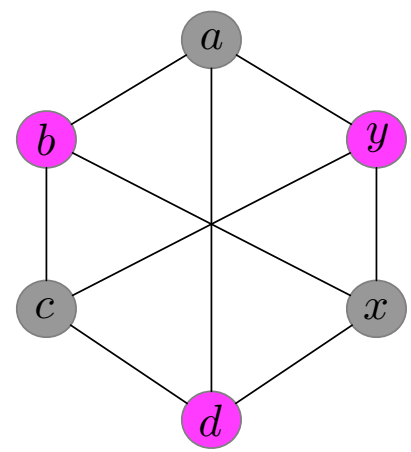
Bipartite Graph:

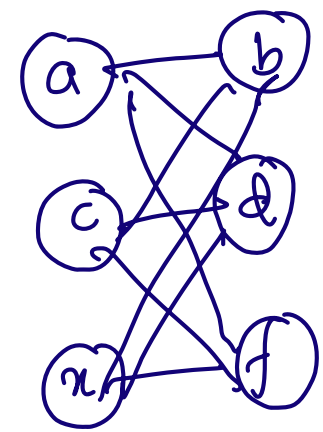
- An undirected graph $G(V, E)$ is said to be bipartite if $V = A \uplus B$ such that for every edge $(u, v) \in E$, $u \in A$ if and only if $v \in B$.

- **Alternate characterization:** An undirected graph is bipartite if it has no cycle with odd number of vertices.

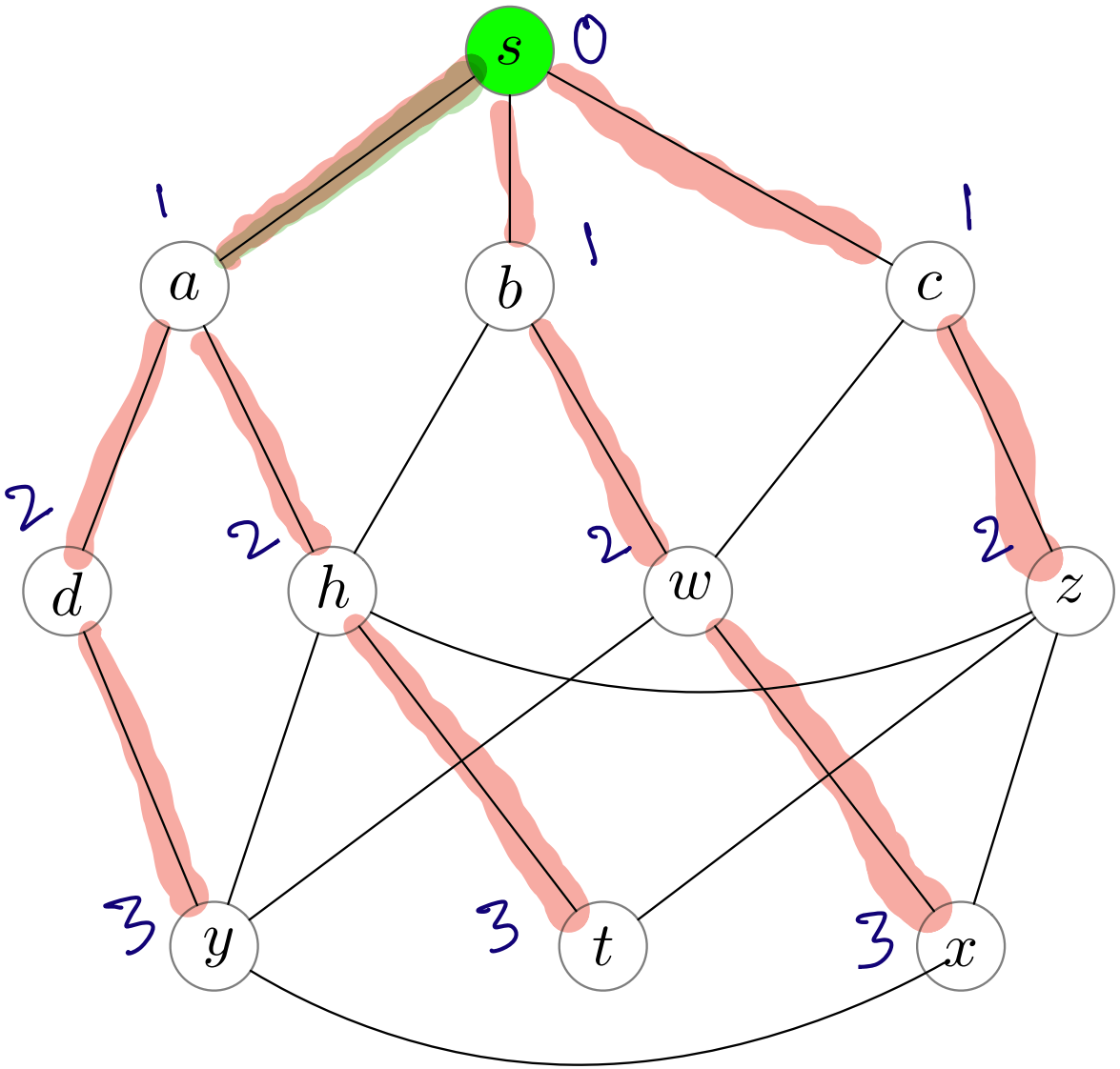
- In other words, a bipartite graph is a graph with no odd cycle.





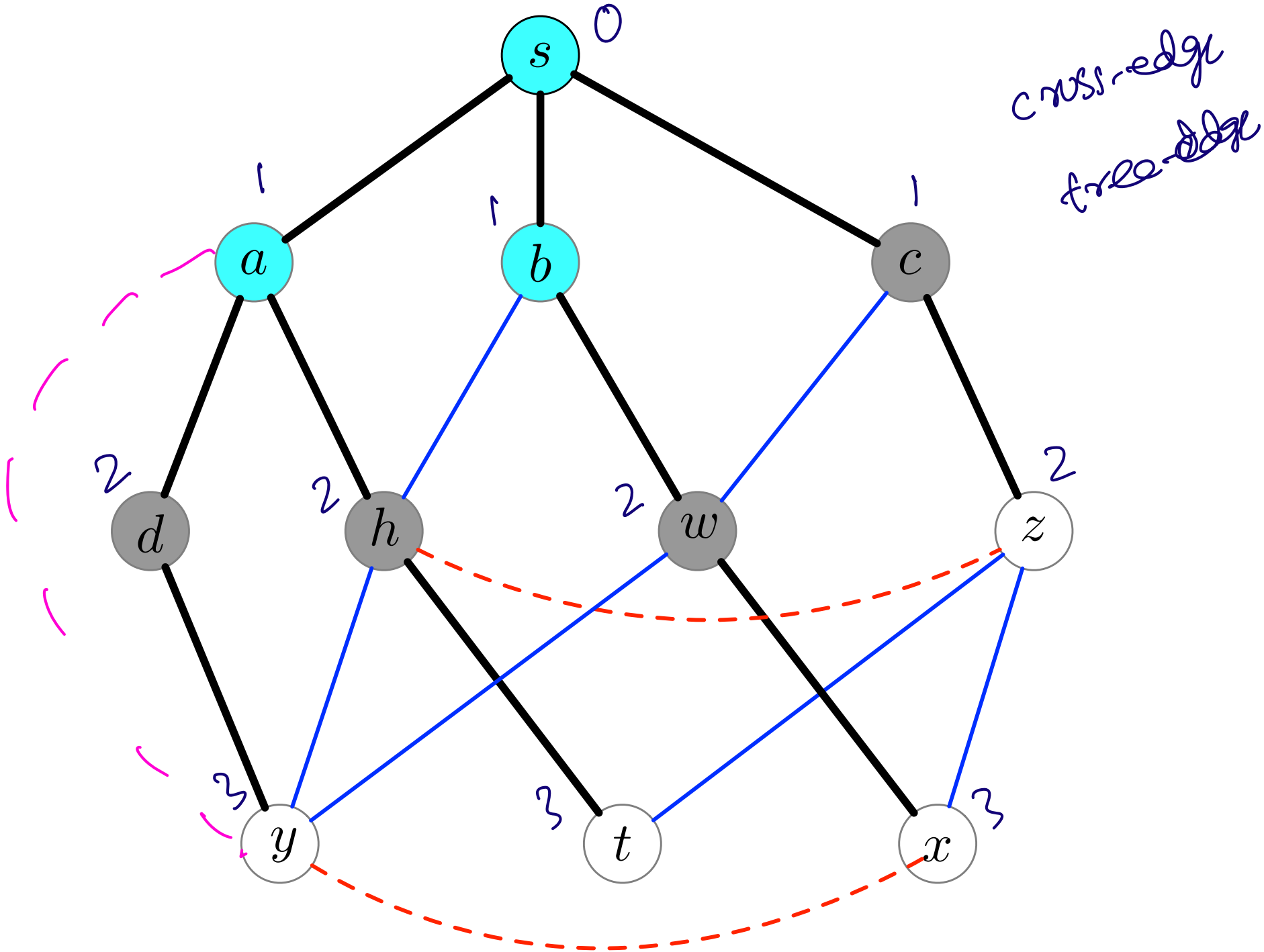


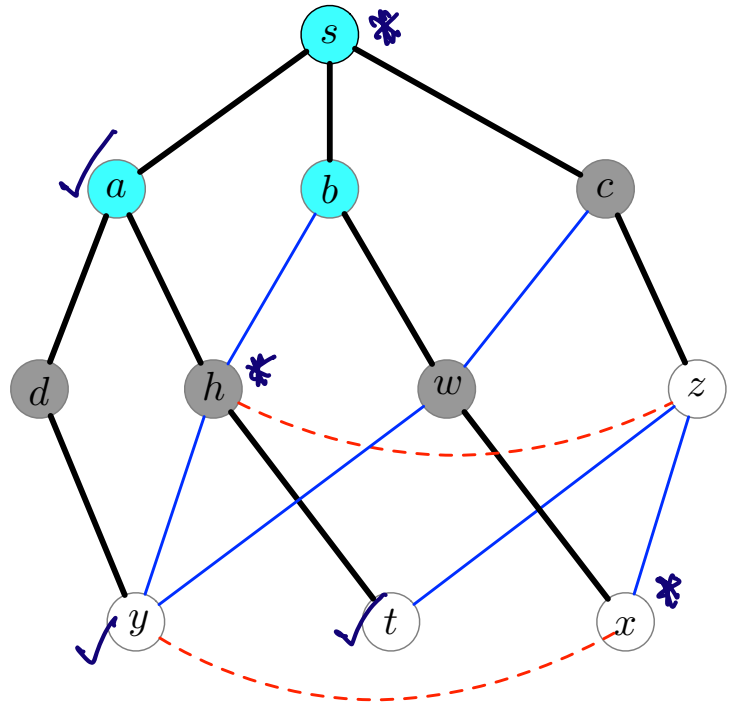
```
BFS(G, s)
for all u ∈ V(G) do
    Color u white;
end
EnQueue(Q, s);
Color u grey; Layer(s) = 0; Initialize queue Q ← ∅; parent(s) = NULL;
while Q is not empty do
    u ← DeQueue(Q);
    for each v ∈ Adj(u) do
        if v is white then
            Color v grey; Layer(v) = Layer(u) + 1;
            parent(v) = u; EnQueue(Q, v);
        end
    end
    Color u black;
end
```


s
0
a
1
b
1
c
1
d
2
h
2
w
2
z
2
y
3
t
3
x
3

Let G be a connected undirected graph.

- $dist(s, u)$ is the number of edges in a shortest path from s to u in G.
- For any edge $(u, v) \in E$, $dist(s, u) \leq dist(s, v) + 1$.
- BFS starting from s terminates with values $Layer(v) \geq dist(s, v)$ for all $v \in V$.
- If G is connected, then $BFS(G, s)$ terminates with $dist(s, v) = Layer(v)$ for all vertices $v \in V$.
- Every edge $(u, v) \in E$ satisfies $Layer(v) \leq Layer(v)$.


s
0
a
1
b
1
c
1
d
2
h
2
w
2
z
2
y
3
t
3
x
3
cross-edge
tree-edge



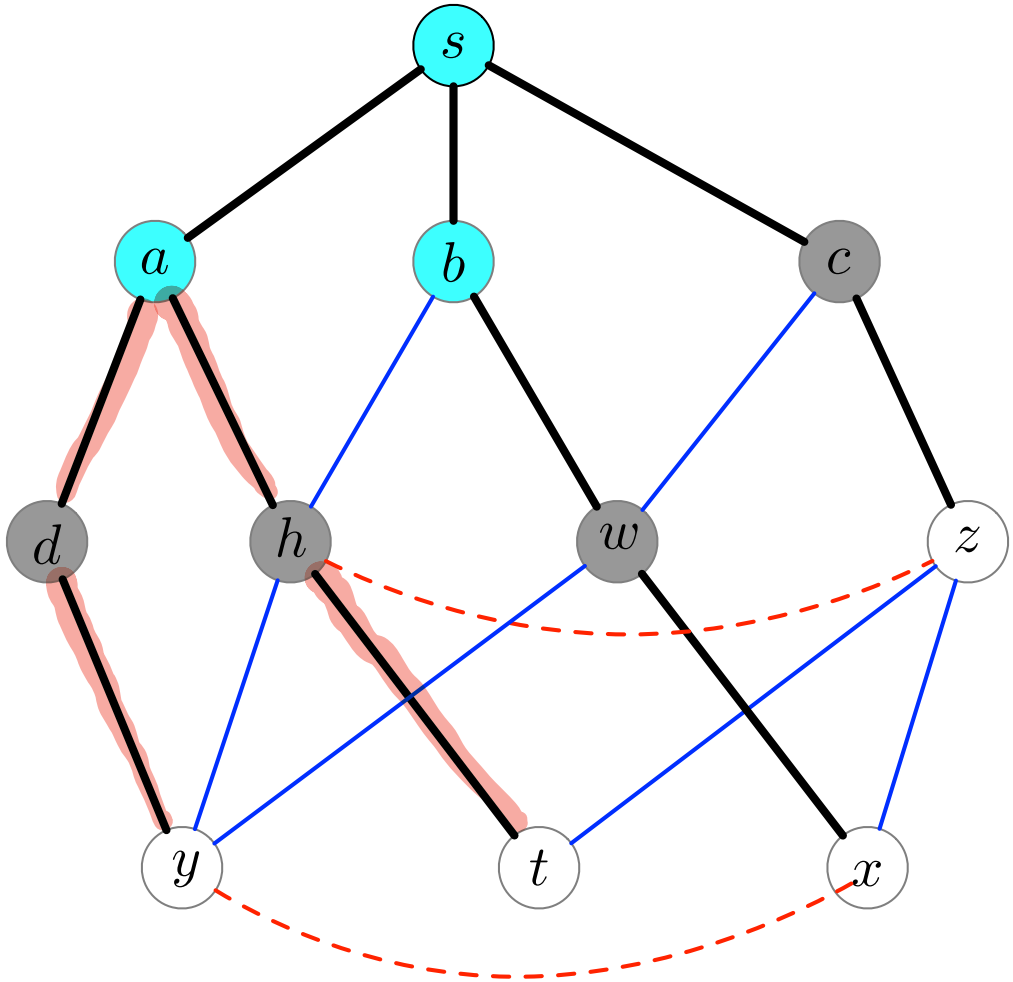
- BFS cannot have back-edge or forward edge.
- **Tree Edge:** An edge (u, v) is a tree-edge if $dist(s, v) = dist(s, v) + 1$. In other words, $Layer(v) = Layer(u) + 1$.
- **Cross Edge:** An edge (u, v) is a cross-edge if $Layer(v) = Layer(u)$ or $Layer(v) = Layer(u) + 1$.

rooted

- **Lowest Common Ancestor (LCA):** Given a tree T, the lowest common ancestor of u and v in T is denoted as $LCA_T(u, v)$.

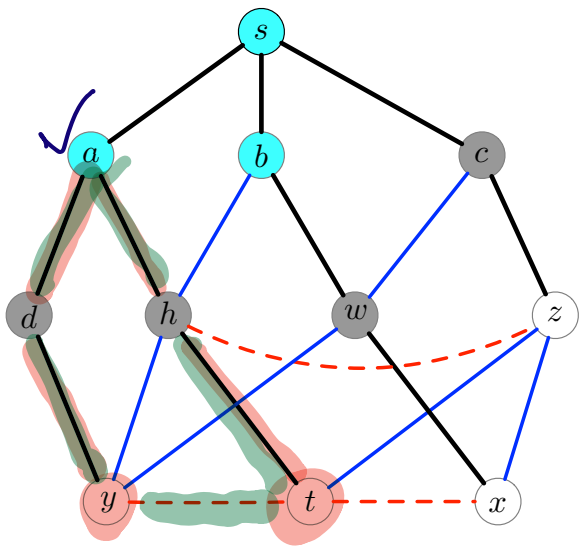
Characterization of Bipartiteness

- An undirected graph G is bipartite if and only if $BFS(G, s)$ does not have any (cross)-edge (u, v) satisfies $Layer(u) = Layer(v)$.

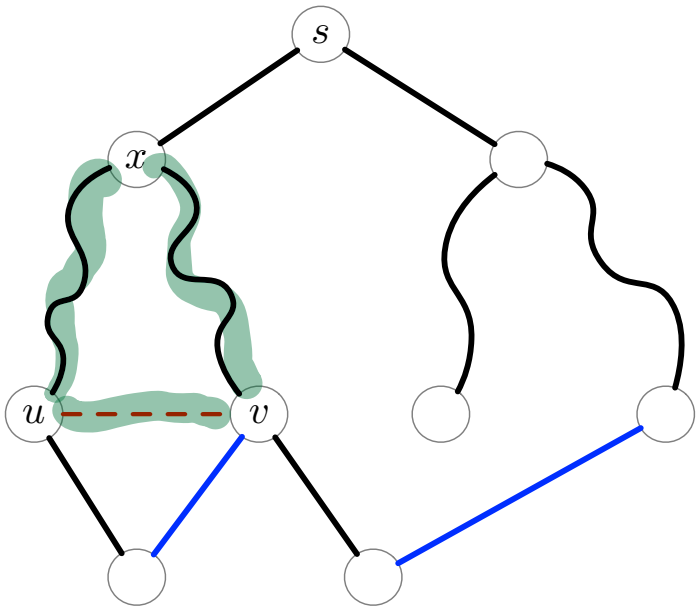


- **Necessary and Sufficient Condition:** An undirected graph G is bipartite if and only if $BFS(G, s)$ does not have any (cross)-edge (u, v) satisfies $Layer(u) = Layer(v)$.

Proof (Sketch):



- $(\Rightarrow)$ Let (y, t) be a cross-edge with $Layer(y) = Layer(t)$.
- Let a be the lowest common ancestor (LCA) of y and t in the BFS-Tree T.
- Then, there is a path $P_{a,y}$ from a to y containing $dist(s, y) - dist(s, a)$ edges.
- There is a path $P_{a,t}$ from a to t containing $dist(s, t) - dist(s, a)$ edges.
- $P_{a,t} \cap P_{a,y}$ are pairwise vertex-disjoint. (except a)
- Consider the number of edges in $E(P_{a,t}) \cup E(P_{a,y}) \cup \{(y, t)\}$.

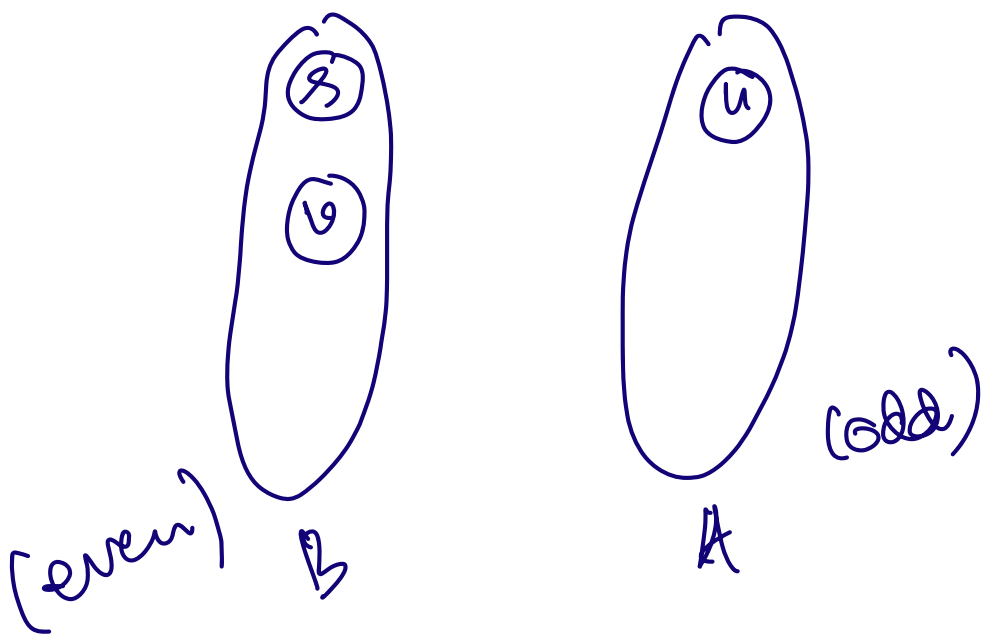


- Let x be the lowest common ancestor (LCA) of u and v in the BFS-Tree T, i.e. $x = LCA_T(u, v)$.
- Then, there is a path $P_{x,u}$ from x to u containing $Layer(u) - Layer(x)$ edges.
- There is a path $P_{x,v}$ from x to v containing $Layer(v) - Layer(x)$ edges.
- $P_{x,u} \cap P_{x,v}$ are pairwise vertex-disjoint.
- The number of edges in $E(P_{x,u}) \cup E(P_{x,v}) \cup \{(u, v)\}$ is $[Layer(u) + Layer(v) - 2Layer(x)] + 1$.
- $P_{x,u} \cup P_{x,v} \cup \{(u, v)\}$ is a cycle with odd number of vertices.

- **Necessary and Sufficient Condition:** An undirected graph G is bipartite if and only if $BFS(G, s)$ does not have any (cross)-edge (u, v) satisfies $Layer(u) = Layer(v)$.

Proof (continued):

- $(\Leftarrow)$ Suppose that $BFS(G, s)$ has no cross-edge (u, v) satisfying $Layer(u) = Layer(v)$.
- Then, construct a bipartition $V(G) = A \uplus B$ as follows.



- **Necessary and Sufficient Condition:** An undirected graph G is bipartite if and only if $BFS(G, s)$ does not have any (cross)-edge (u, v) satisfies $Layer(u) = Layer(v)$.

Proof (continued):

- $(\Leftarrow)$ Suppose that $BFS(G, s)$ has no cross-edge (u, v) satisfying $Layer(u) = Layer(v)$.
- Then, construct a bipartition $V(G) = A \uplus B$ as follows.
- Invoke $BFS(G, s)$ and compute $Layer(v)$ for all $v \in V$.
- Initialize $A, B \leftarrow \emptyset$. If $Layer(v)$ is odd, then $A \leftarrow A \cup \{v\}$. If $Layer(v)$ is even, then $B \leftarrow B \cup \{v\}$.
- There is no edge (u, v) such that $Layer(u) = Layer(v)$.
- Every edge (u, v) satisfies $Layer(u) = Layer(v) + 1$.
- There is no edge (u, v) such that $Layer(u) \geq Layer(v) + 2$.
- Hence, any edge (u, v) of G satisfies $u \in A$ if and only if $v \in B$.

```
BFS(G, s)
For all v ∈ V, color v white;
Color u grey; Layer(s) = 0; Initialize queue Q ← ∅; parent(s) = NULL;
while Q is not empty do
    u ← DeQueue(Q);
    for each v ∈ Adj(u) do
        if v is white then
            Color v grey; Layer(v) = Layer(u) + 1;
            parent(v) = u; EnQueue(Q, v);
        end
        else
            if Layer(u) = Layer(v) then
                Output there is an odd cycle;
            end
        end
    end
    Color u black;
end
Output G is bipartite graph.
```

Algorithm 2

- **Stage 1:** Invoke $BFS(G, s)$ and obtain compute $Layer(v)$ for all $v \in V$.
- **Stage 2:** For every edge $(u, v) \in E$, check if $Layer(u) = Layer(v)$.
- If for all $(u, v) \in E$, it holds that $Layer(u) \neq Layer(v)$, then output that G is bipartite graph.
- Otherwise, there is $(u, v) \in E$ such that $Layer(u) = Layer(v)$. Output that G is not bipartite.

- Section 3.4 of Algorithm Design (Kleinberg and Tardos).

Analysis of MST algorithms

Debarka Sengupta

Minimum Spanning Tree

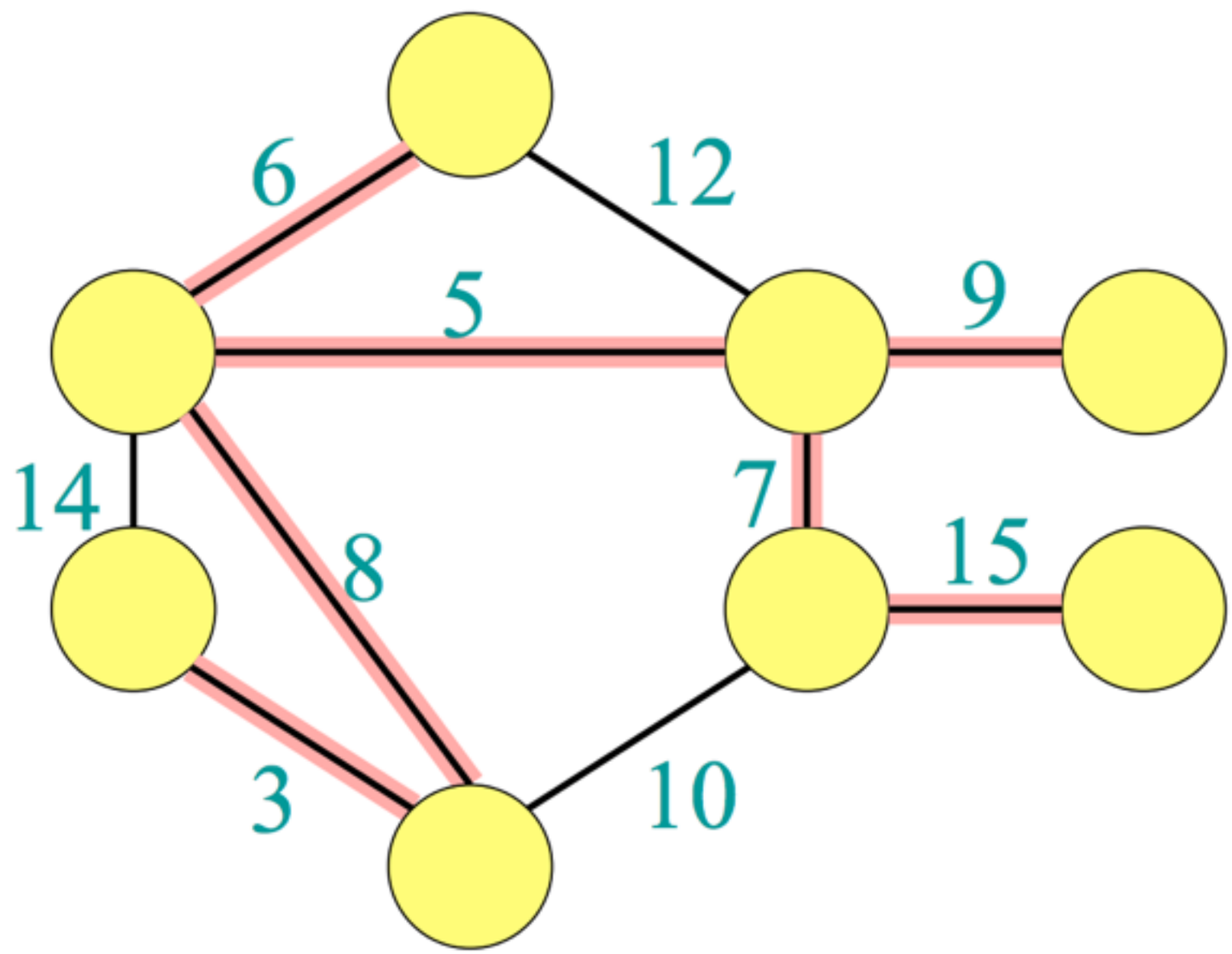
Input: A connected undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$.

- For now, assume all edge weights are distinct.

Output: A ***spanning tree*** T — a tree that connects all vertices — of minimum weight:

$$w(T) = \sum_{(u,v) \in T} w(u, v).$$

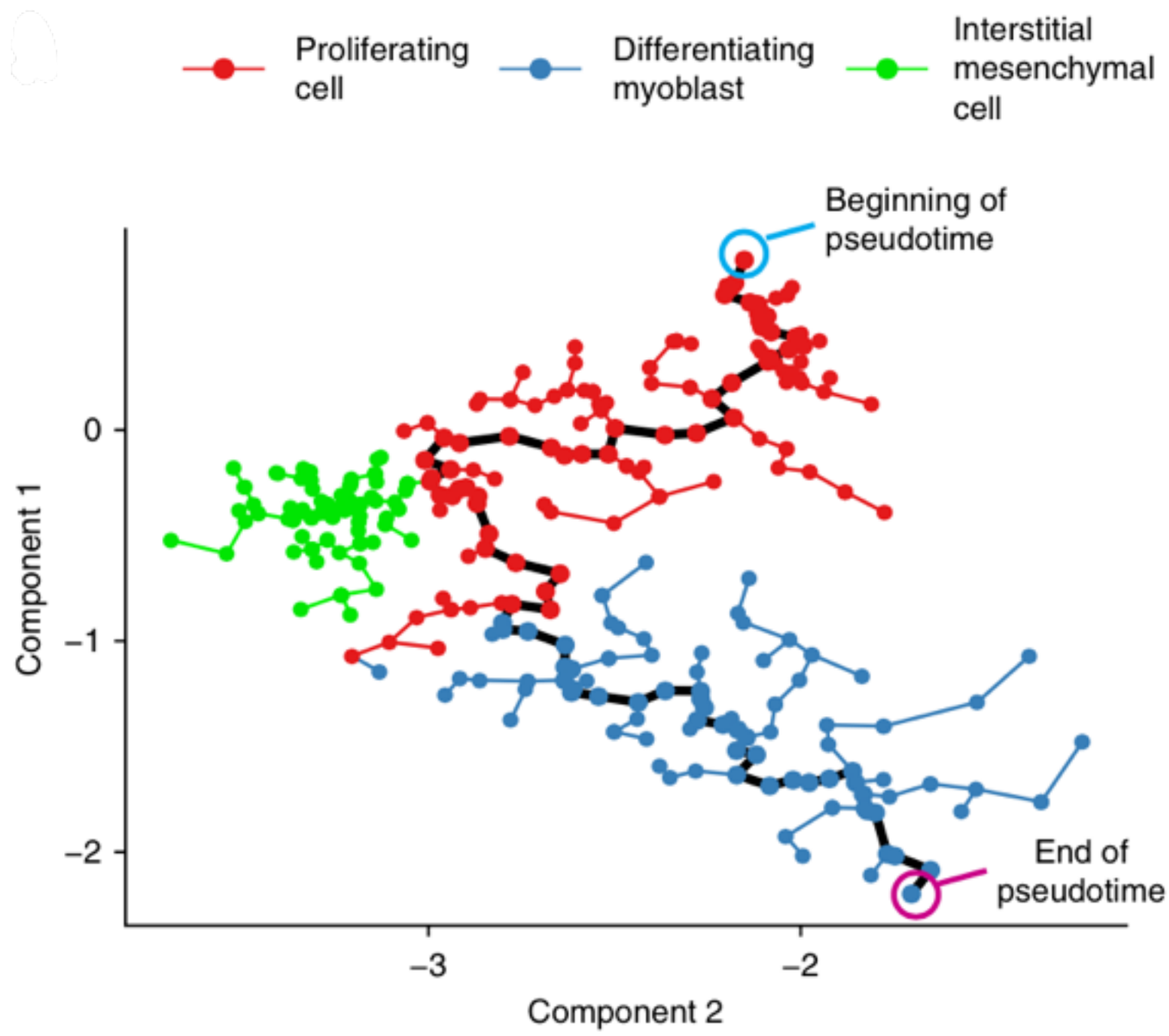
Example



Applications

- Telephone network
- Approximation of hard problems such as TSP
- Pseudo temporal ordering

Pseudo-temporal ordering



Cut

A cut of a graph is a partition of its vertices into two nonempty disjoint sets. A crossing edge of a cut is an edge that connects a vertex in one set with a vertex in the other.

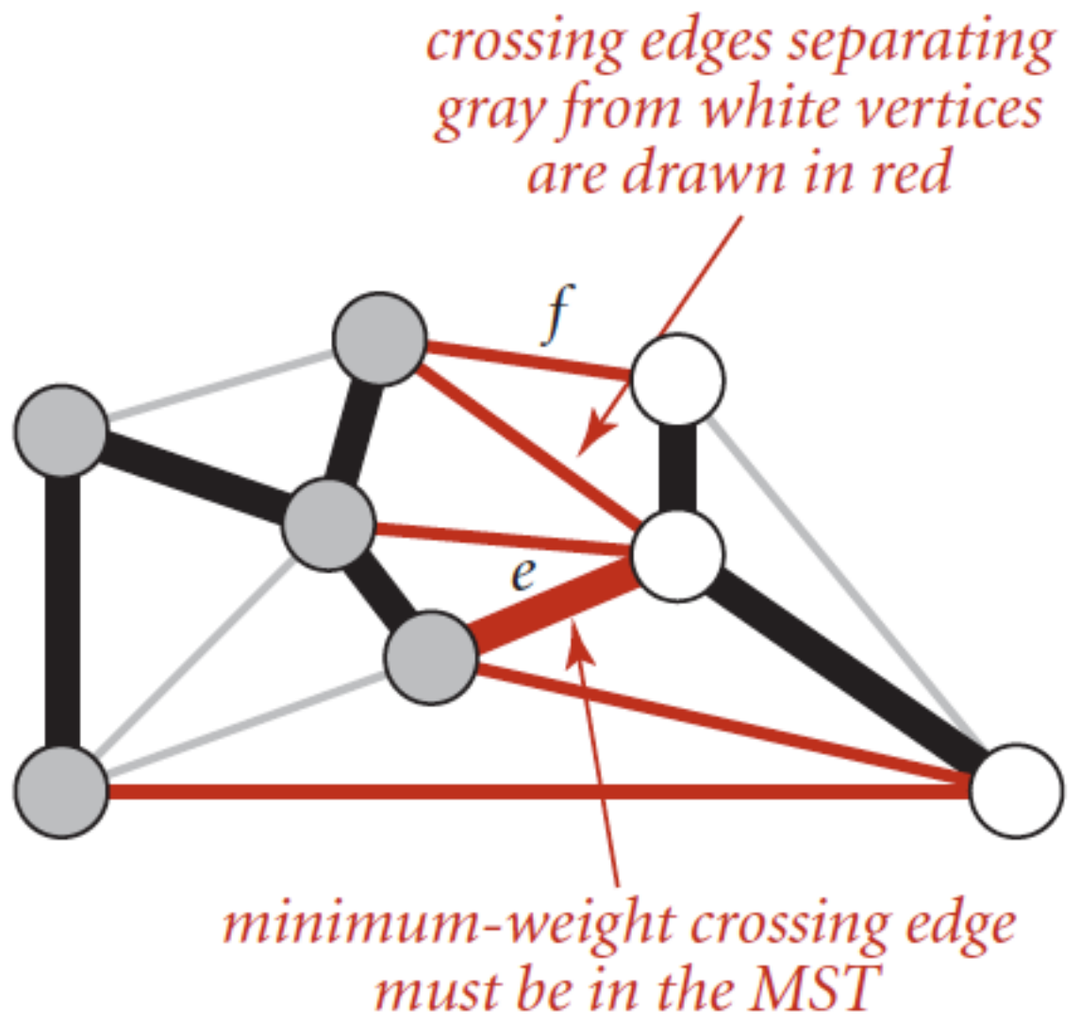
Cut-set

Collection of edges which connect vertices residing across vertex sets formed due to a cut

Cut-set and crossing edge

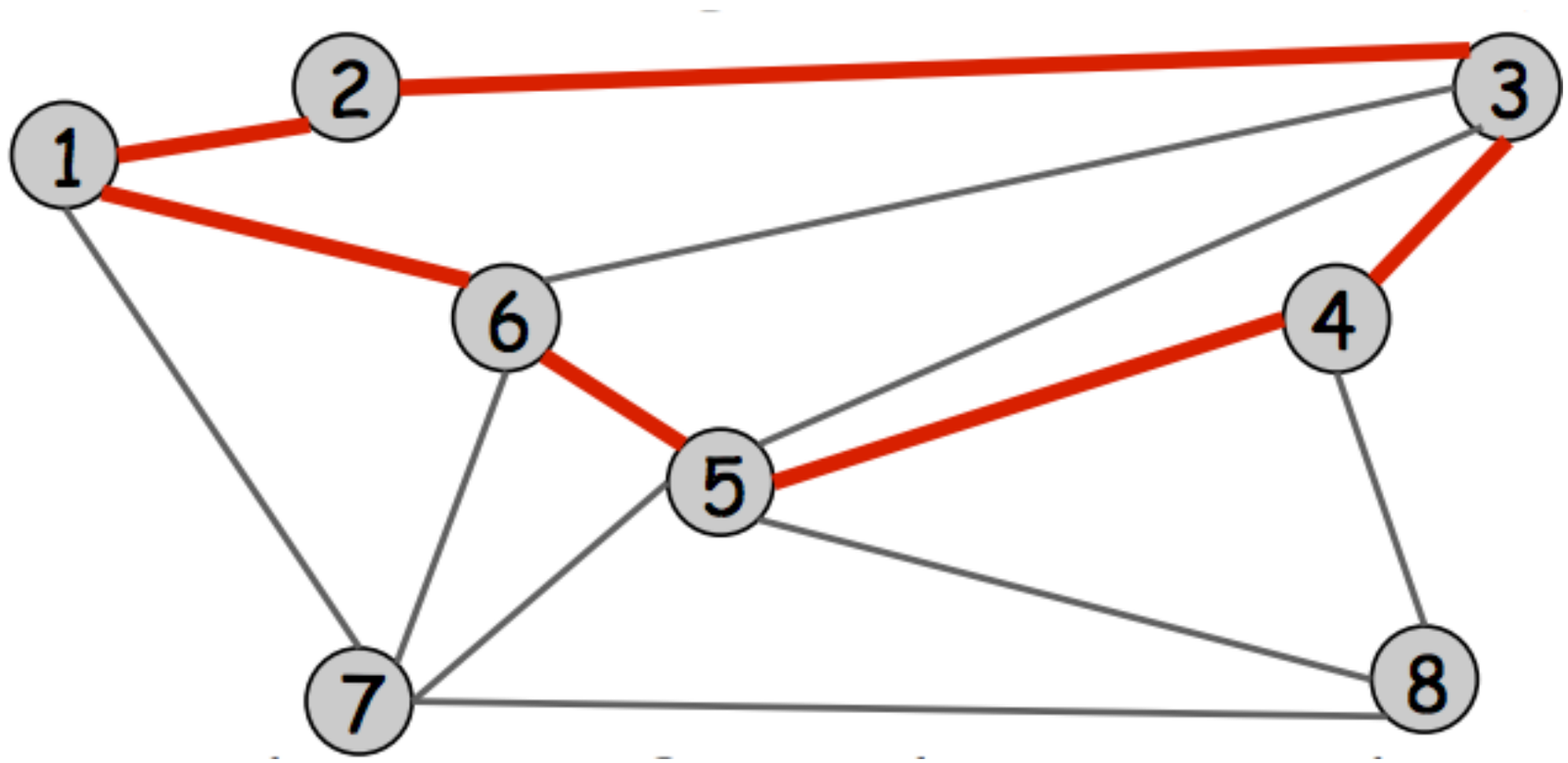
Subset of edges which connect vertices residing across vertex sets formed due to a cut. Each edge is called a crossing edge. Crossing edge with minimum weight is called light edge.

Illustration



Cut property

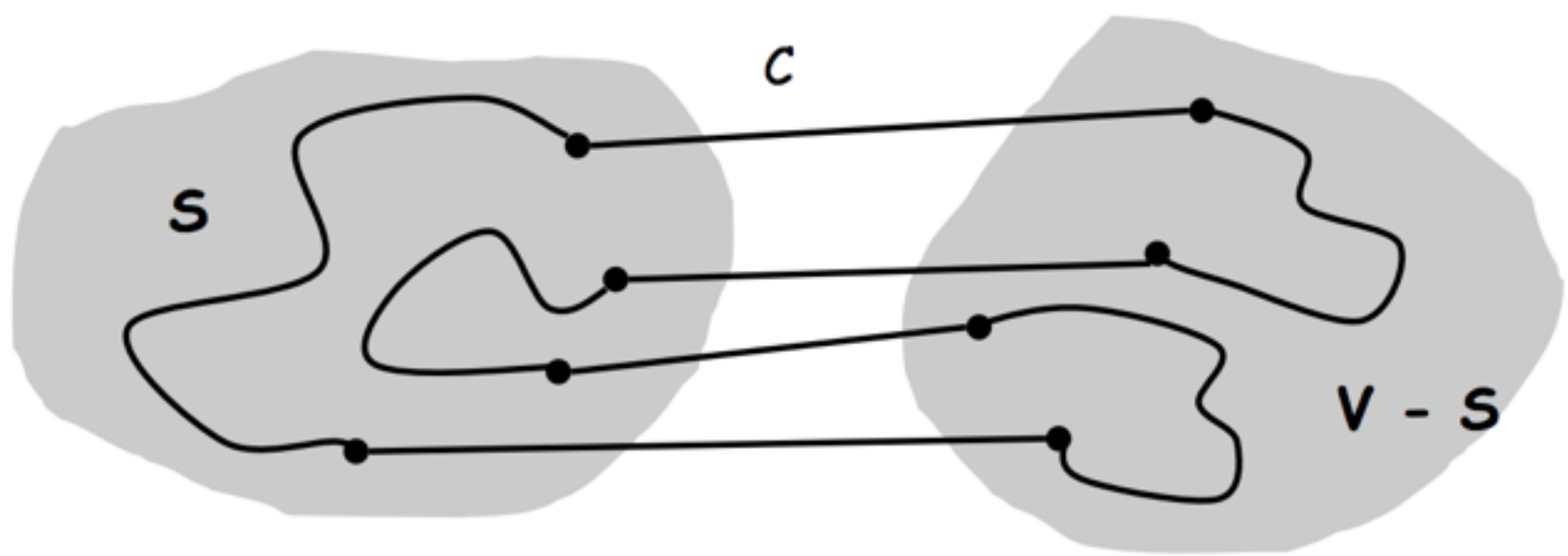
Cycle



Cycle C = (1,2),(2,3),(3,4),(4,5),(5,6),(6,1)

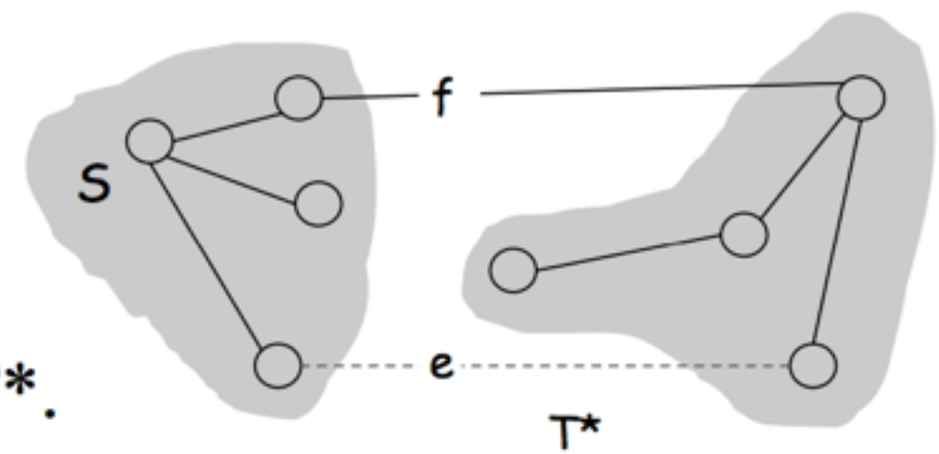
Cycle Cut intersection

- **Claim.** A cycle and a cutset intersect in an even number of edges.
- **Proof:** A cycle has to leave and enter the cut the same number of times.



Cut property

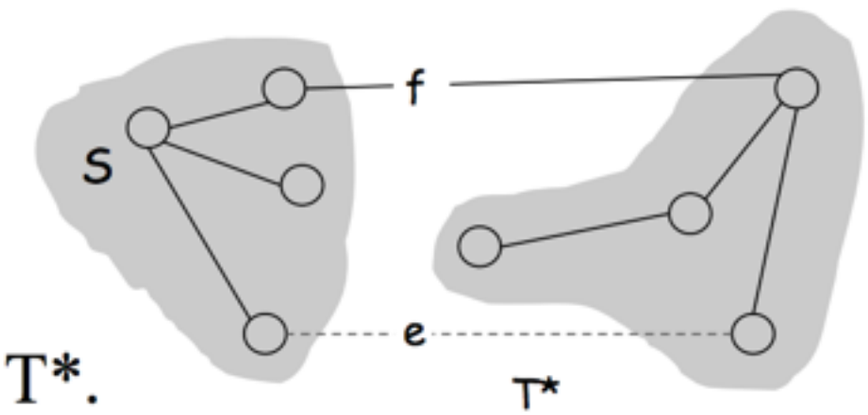
Cut property: Let S be a subset of nodes. Let e be the min weight edge with exactly one endpoint in S. Then the MST T* contains e.



- **Proof:** (exchange argument)
  - Suppose e does not belong to T*.
  - Adding e to T* creates a cycle C in T*.
  - Edge e is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say f, that is in both C and D.
  - $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
  - Since $c_e < c_f$, cost(T') < cost(T*). Contradiction. ▪

Cycle property

Cycle property: Let C be a cycle in G. Let f be the max weight edge in C. Then the MST T* does not contain f.



- **Proof:** (exchange argument)
  - Suppose f belongs to T*.
  - Deleting f from T* creates a cut S in T*.
  - Edge f is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say e, that is in both C and D.
  - $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
  - Since $c_e < c_f$, cost(T') < cost(T*). Contradiction. ▪

Generic greedy template

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until V - 1 edges are colored black.

Special cases of the greedy template

- **Kruskal's:** Start with $T = \phi$. Consider edges in ascending order of weights. Insert edge e in T unless doing so would create a cycle.

- **Reverse-Delete:** Start with $T = E$. Consider edges in descending order of weights. Delete edge e from T unless doing so would disconnect T.

- **Prim's:** Start with some root node s. Grow a tree T from s outward. At each step, add to T the cheapest edge e with exactly one endpoint in T.

Prim's Algorithm: Correctness

- Induction
- The induction hypothesis will be that after each iteration, the tree **T** is a subgraph of some minimum spanning tree **M**
- True at start since **T** is just a single node - no edge
- Now suppose that at some point in the algorithm we have **T** which a subgraph of **M**, and Prim's algorithm tells us to add the edge **e**

Prim's Algorithm: Correctness

- Two cases might arise
  - **e** is in **M** - then apparently there's no problem
  - **e** is not in **M** - Then according to the cut property there's a cross edge **e'** that belongs to **M**
  - But going by Prim's algorithm **e'** cannot be less than **e** in terms of weight
  - Now, at most **e** can be equal to **e'**. In that case either can be dropped retaining the MST property

Kruskal's Algorithm: Correctness

- Similar to Prim’s algorithm
- The idea is that every time an edge is considered without giving rise to cycle, it’s absence does not make the tree more promising
- Write the proof yourself

Dijkstra's algorithm

By Laksman Veeravagu and Luis Barrera

- Important Definitions
  - Path Finding Algorithms
    - DFS, BFS, Dijkstra's, Bellman-Ford, etc.
  - Shortest ***Unweighted*** Path Finding Algorithm
    - BFS
  - Shortest ***Weighted*** Path Finding Algorithm
    - Dijkstra's Algorithm, Bellman-Ford, etc.
- Applications
  - Google maps, network routing, etc.

The author: Edsger Wybe Dijkstra

"Computer Science is no more about computers than astronomy is about telescopes."

http://www.cs.utexas.edu/~EWD/

Edsger Wybe Dijkstra

- May 11, 1930 – August 6, 2002

- Received the 1972 A. M. Turing Award, widely considered the most prestigious award in computer science.

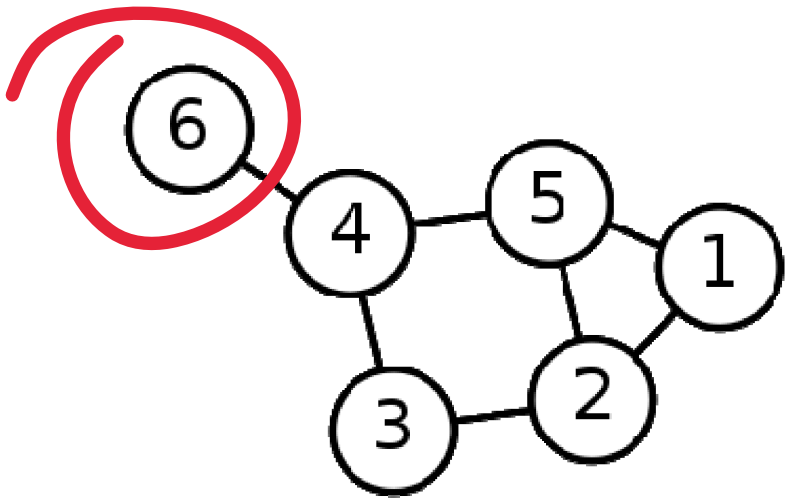
- The Schlumberger Centennial Chair of Computer Sciences at The University of Texas at Austin from 1984 until 2000

- Made a strong case against use of the GOTO statement in programming languages and helped lead to its deprecation.

- Known for his many essays on programming.

Single-Source Shortest Path Problem

<u>Single-Source Shortest Path Problem</u> - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

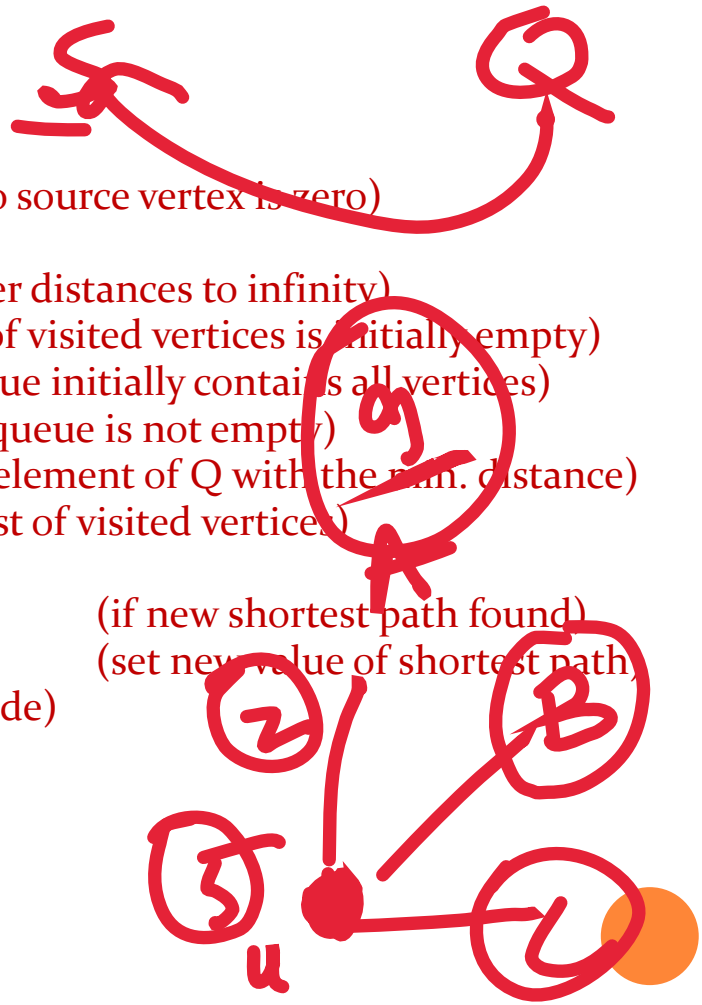
Approach: Greedy

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

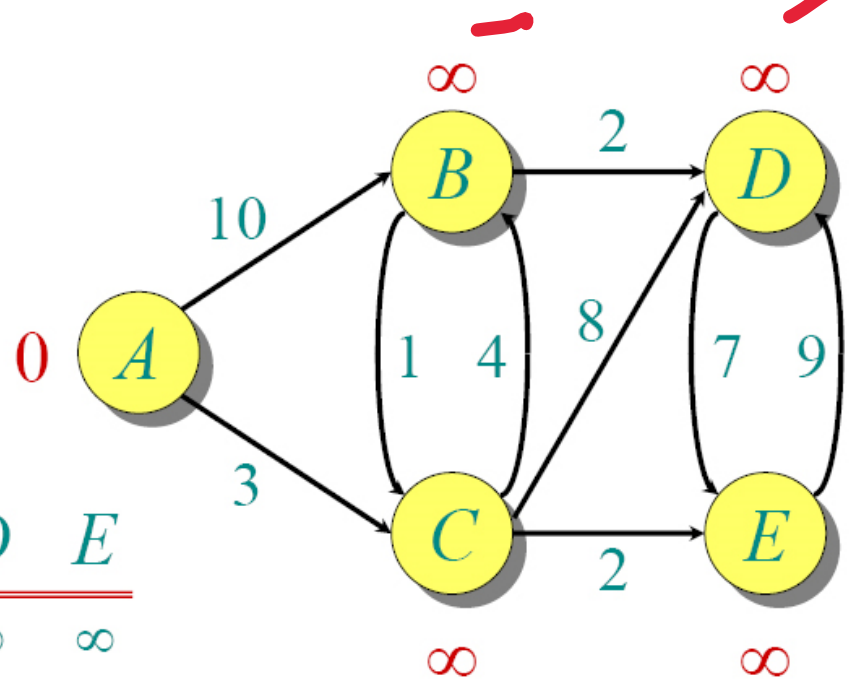
Dijkstra's algorithm - Pseudocode

```
dist[s] ←0                                  (distance to source vertex is zero)
for  all v ∈ V–{s}
        do  dist[v] ←∞                      (set all other distances to infinity)
S←∅                                         (S, the set of visited vertices is initially empty)
Q←V                                         (Q, the queue initially contains all vertices)
while Q ≠∅                                  (while the queue is not empty)
do   u ← mindistance(Q,dist)                (select the element of Q with the min. distance)
     S←S∪{u}                                (add u to list of visited vertices)
        for all v ∈ neighbors[u]
              do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                       then      d[v] ←d[u] + w(u, v)       (set new value of shortest path)
                          (if desired, add traceback code)
return dist
```



Dijkstra Animated Example

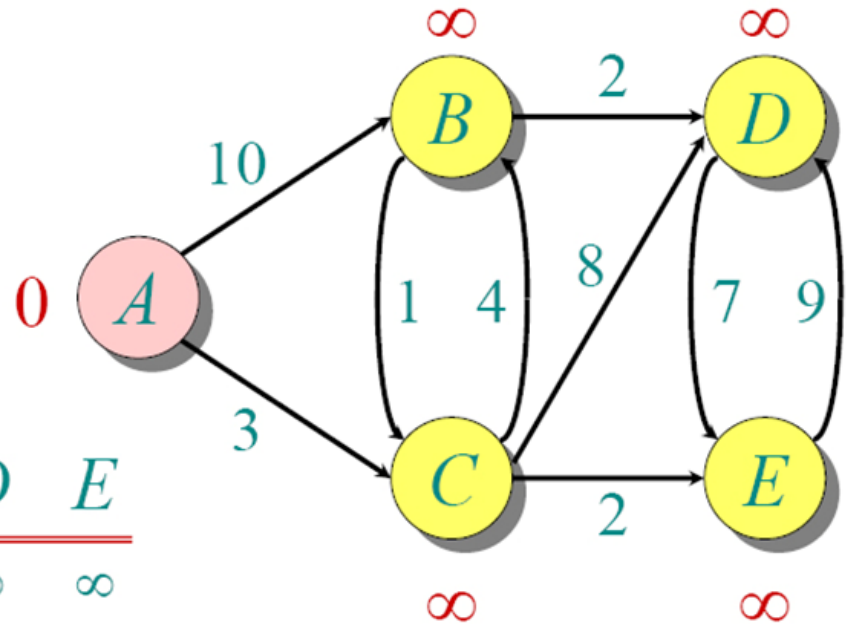
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

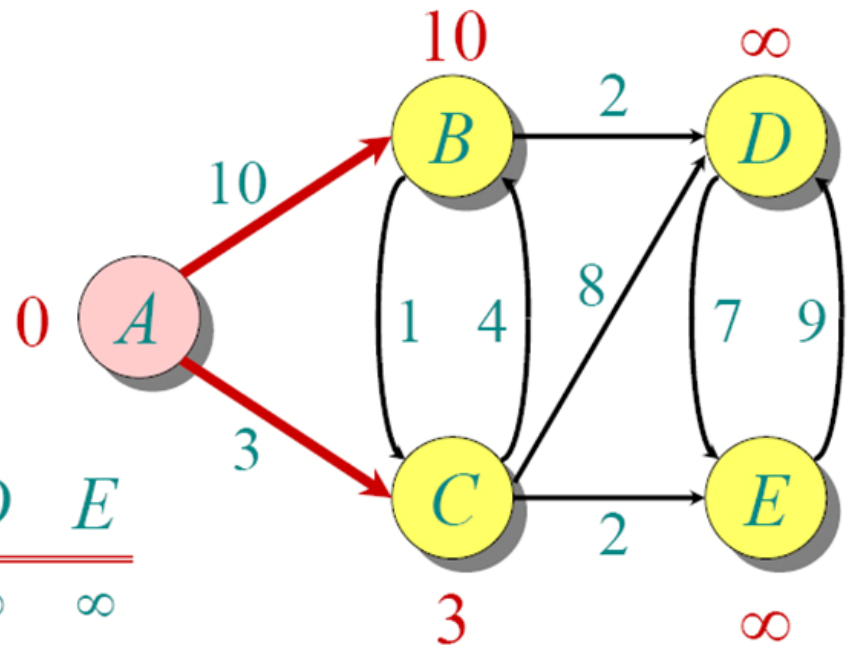
S: {}

Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

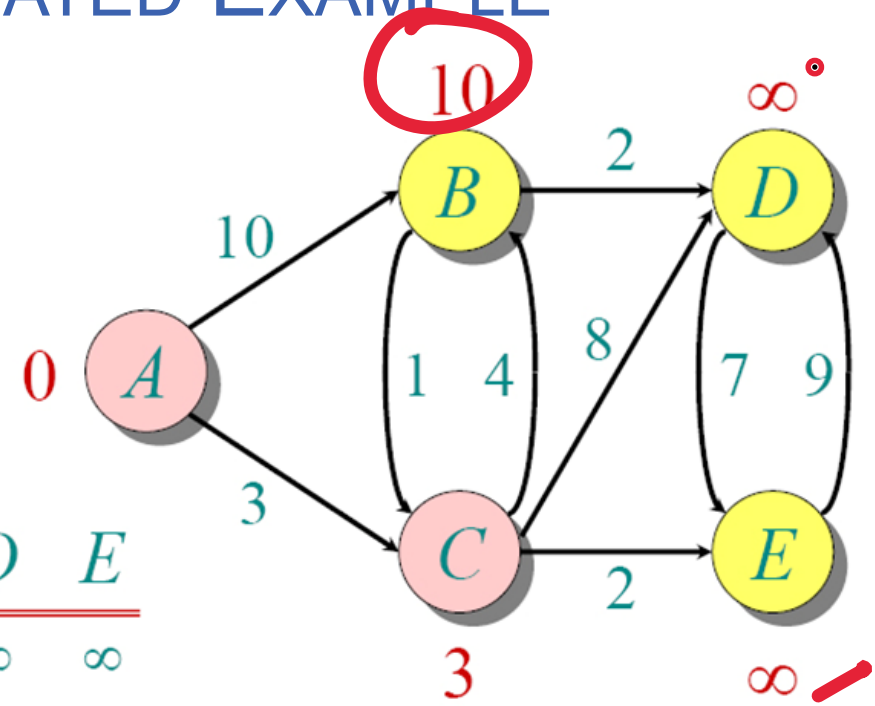
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

Dijkstra Animated Example

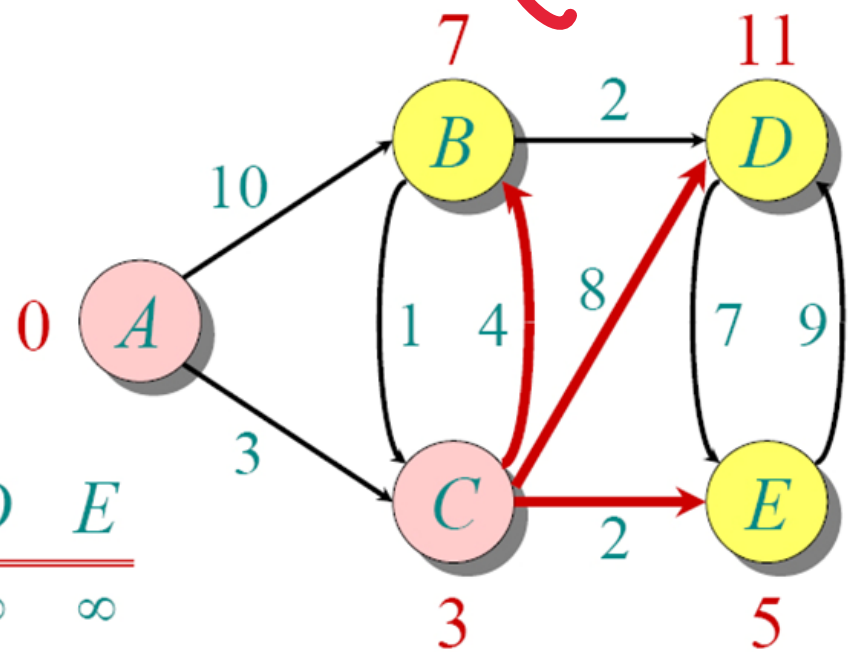


Q: A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞

S: { A, C }

Dijkstra Animated Example

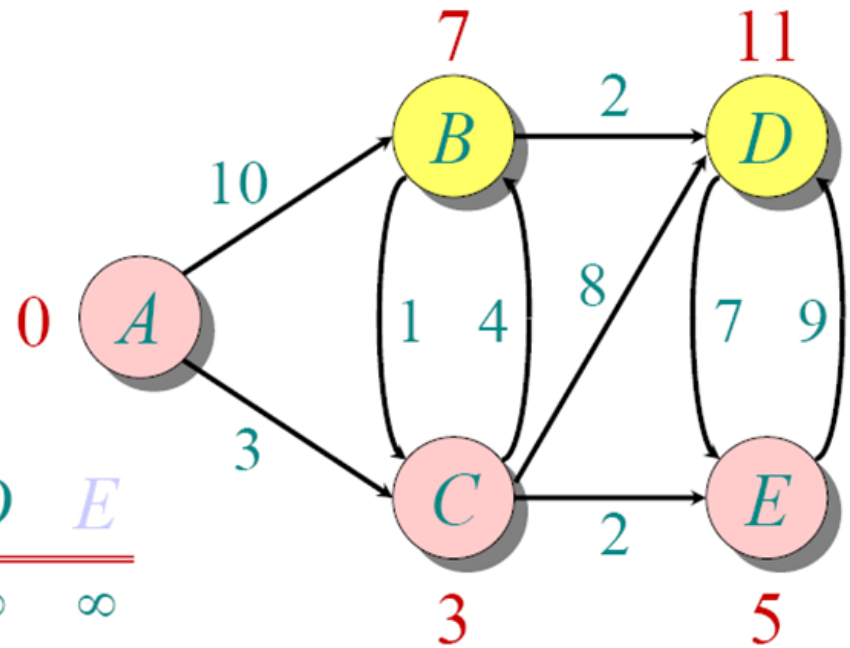
(10)



Q: A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5

S: { A, C }

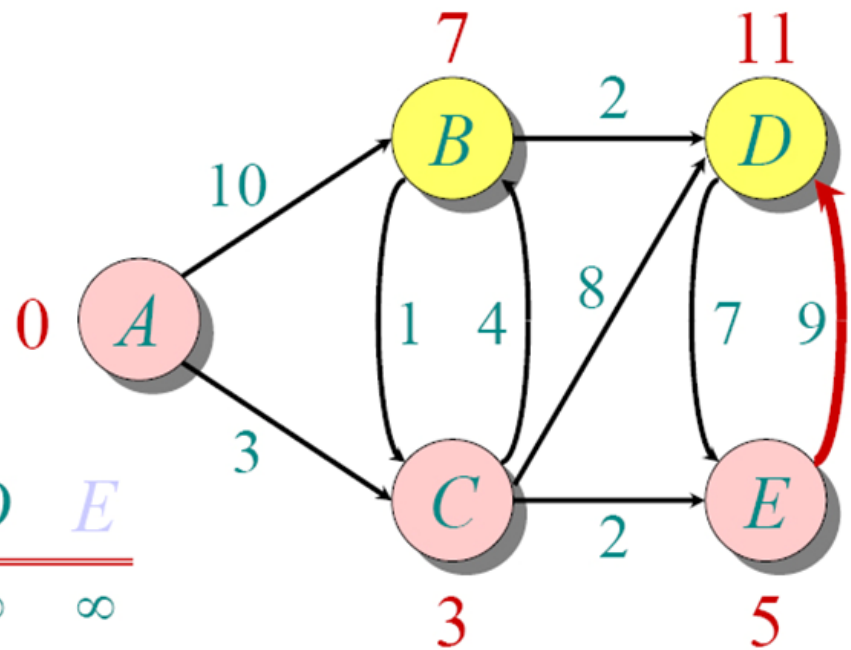
Dijkstra Animated Example



Q: A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5

S: { A, C, E }

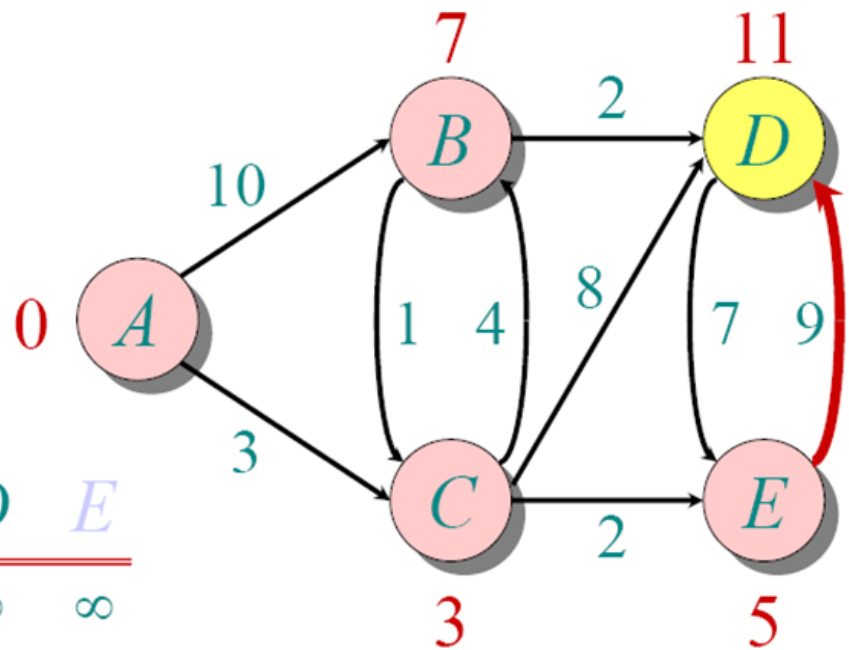
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

Dijkstra Animated Example

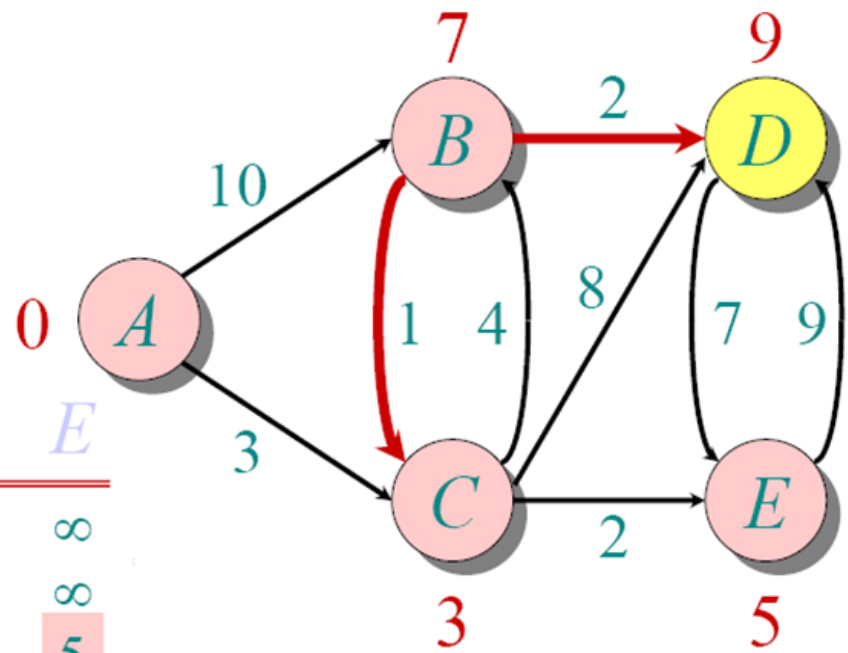


Q: A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	

S: { A, C, E, B }

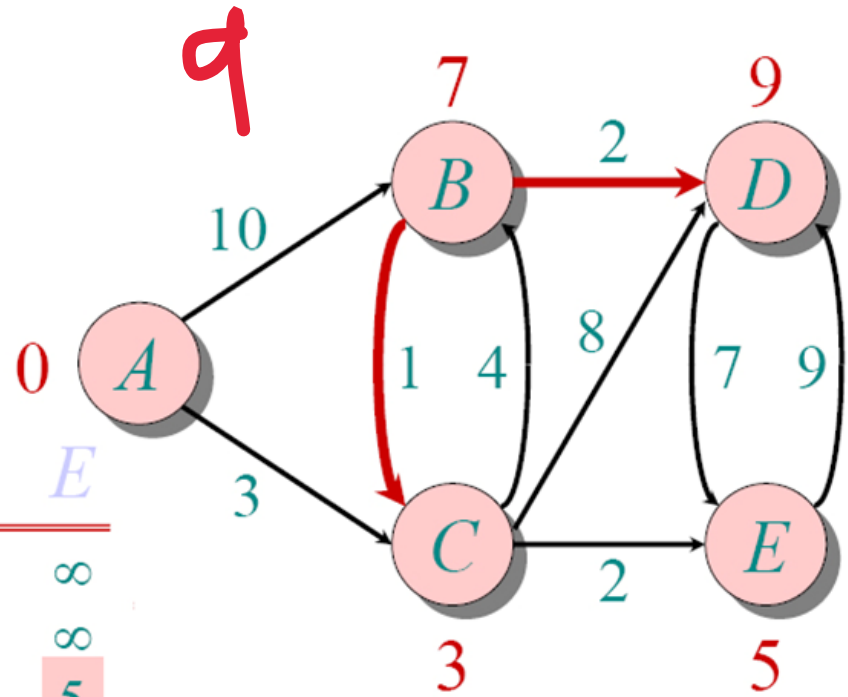
Dijkstra Animated Example

Q: A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	



S: { A, C, E, B }

Dijkstra Animated Example



Q: A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	

S: { A, C, E, B, D }

Implementations and Running Times

The simplest implementation is to store vertices in an array or linked list. This will produce a running time of

O(|V|^2 + |E|)

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of

$O((|E|+|V|) \log |V|)$

Dijkstra's Algorithm - Why It Works

- As with all greedy algorithms, we need to make sure that it is a correct algorithm (e.g., it *always* returns the right solution if it is given correct input).

- A formal proof would take longer than this presentation, but we can understand how the argument works intuitively.

- If you can't sleep unless you see a proof, see the second reference or ask us where you can find it.

Dijkstra's Algorithm - Why It Works

- To understand how it works, we'll go over the previous example again. However, we need two mathematical results first:

- **Lemma 1**: Triangle inequality
  If $\delta(u,v)$ is the shortest path length between u and v,
  $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$
- **Lemma 2**:
  The subpath of any shortest path is itself a shortest path.

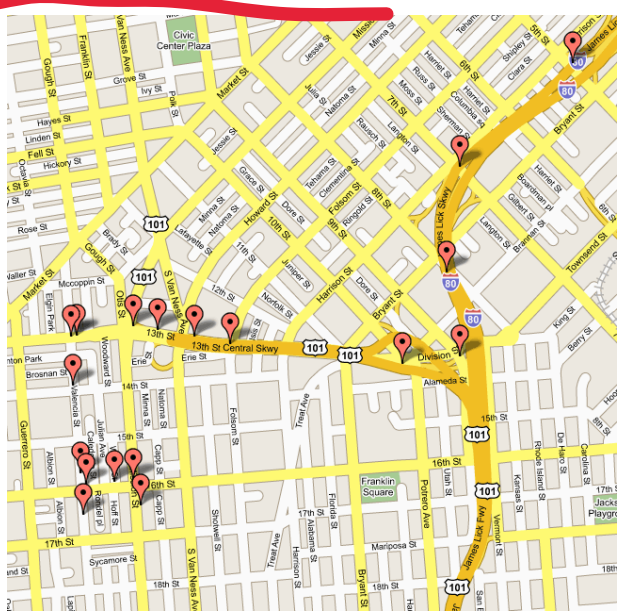
- The key is to understand why we can claim that anytime we put a new vertex in S, we can say that we already know the shortest path to it.
- Now, back to the example…

Dijkstra's Algorithm - Why use it?

- As mentioned, Dijkstra's algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex u to every vertex using Dijkstra's as it is to calculate the shortest path to some particular vertex v.
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra's algorithm.

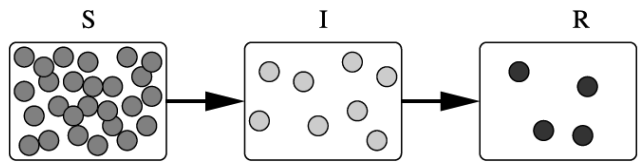
Applications of Dijkstra's Algorithm

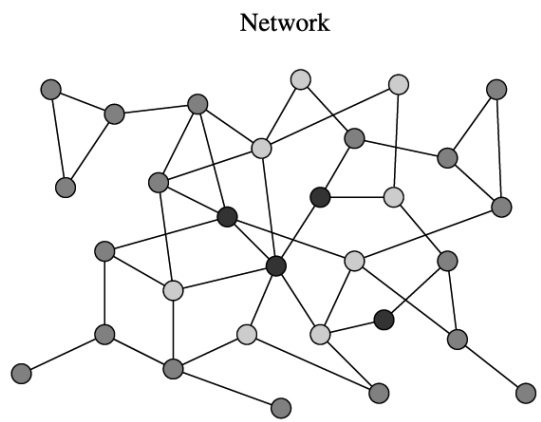
- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems


Civic Center Plaza
Franklin St
Van Ness Ave
Market St
Hayes St
Fell St
Oak St
Grove St
Ivy St
Rose St
Gough St
Mission St
Howard St
Folsom St
Harrison St
Bryant St
Brannan St
Townsend St
King St
Berry St
Embarcadero
Jackson St
Pacific Ave
Broadway
Columbus Ave
Kearny St
Montgomery St
Sansome St
Battery St
Front St
Drumm St
Davis St
Leavenworth St
Jones St
Taylor St
Mason St
Powell St
Stockton St
Grant Ave
Kearny St
Dupont St
Polk St
Larkin St
Van Ness Ave
Franklin St
Gough St
Octavia St
Laguna St
Buchanan St
Webster St
Fillmore St
Steiner St
Pierce St
Scott St
Divisadero St
Broderick St
Baker St
Lyon St
Presidio Ave
Walker St
Mission St
Valencia St
Guerrero St
Dolores St
San Jose Ave
Geneva Ave
Alemany Blvd
Bayshore Blvd
Cesar Chavez St
Army St
Potrero Ave
Indiana St
Illinois St
Third St
Fourth St
Kansas St
Rhode Island St
Connecticut St
Arkansas St
Texas St
Pennsylvania Ave
Minnesota St
Iowa St
Michigan St
Delaware St
Maryland St
Wisconsin St
Nevada St
Idaho St
Utah St
Oregon St
Washington St
Alabama St
Florida St
Mariposa St
Cesar Chavez St
Sycamore St
Franklin Square
Jackson Playground
Mission Bay Blvd North
Mission Bay Blvd South
Third St
Fourth St
Channel St
Terry A Francois Blvd
Illinois St
Pier 70
Pier 80
Pier 90-96
Pier 94
Pier 92
Pier 96
Pier 98
Pier 100
Pier 102
Pier 104
Pier 106
Pier 108
Pier 110
Pier 112
Pier 114
Pier 116
Pier 118
Pier 120
Pier 122
Pier 124
Pier 126
Pier 128
Pier 130
Pier 132
Pier 134
Pier

Applications of Dijkstra's Algorithm

- One particularly relevant this week: epidemiology
- Prof. Lauren Meyers (Biology Dept.) uses networks to model the spread of infectious diseases and design prevention and response strategies.
- Vertices represent individuals, and edges their possible contacts. It is useful to calculate how a particular individual is connected to others.
- Knowing the shortest path lengths to other individuals can be a relevant indicator of the potential of a particular individual to infect others.





References

- Dijkstra's original paper:
  E. W. Dijkstra. (1959) *A Note on Two Problems in Connection with Graphs*. Numerische Mathematik, 1. 269-271.
- MIT OpenCourseware, 6.046J Introduction to Algorithms. < http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/CourseHome/> Accessed 4/25/09
- Meyers, L.A. (2007) Contact network epidemiology: Bond percolation applied to infectious disease prediction and control. *Bulletin of the American Mathematical Society* **44**: 63-86.
- Department of Mathematics, University of Melbourne. *Dijkstra's Algorithm*. <http://www.ms.unimelb.edu.au/~moshe/620-261/dijkstra/dijkstra.html > Accessed 4/25/09

Runtime Complexity

Dijkstra(G, s)

```
for each vertex v ∈ G: v.d ← ∞, v.π ← nil
s.d ← 0
add every vertex in G to Q
while Q not empty
        u ← ExtractMin(Q)
        for each vertex v such that u → v
                Relax(u, v)
```

$(|V|)$

$\log|V|$

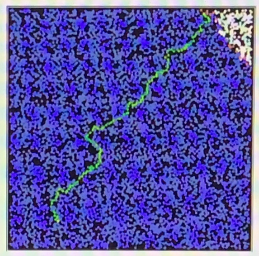
Dijkstra-Complexity

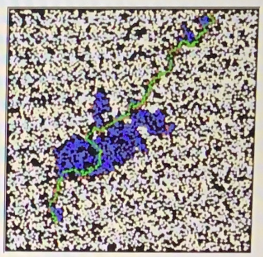
- Dijkstra's algorithm is structurally identical to BFS.
- However, it is slower because the priority queue is computationally more demanding than the constant-time pop and push of BFS.

- Runtime analysis using a binary heap:
  - BuildPriorityQueue takes O(|V|).
  - ExtractMin and Insert are executed |V| times each – once for each node, thus it takes O(|V|log|V|).
  - In the worst case, DecreaseKey can be executed for every edge, thus |E| times, giving O(|E|log|V|).
- Overall we get O((|V|+|E|)log|V|)

$|V|\log|V| + |E|\log|V|$

- Dijkstra's with a heuristic (*h*)
  - *h* is the "estimated movement cost to move from one node to the final destination"
- Calculating *h* can be exact or estimated with distances
  - Manhattan, Diagonal, and Euclidean distance metrics are popular estimations for *h*
- Runtime Complexity: *O*(E)
- When to use?
  - One Source → One Destination

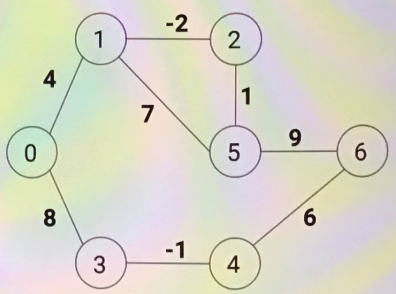




$(E+V)\log_2 V$

$\log V$ $\quad$ V

- Dijkstra's algorithm doesn't work for graphs with negative weight edges
  - But Bellman-Ford's does
- For every node V, all edges are examined and an array of shortest distances to each node is built
- Runtime Complexity: *O*(VE)
- When to use?
  - One Source → ALL Destination Nodes
    - (When graph has negative weight edges)



- "All Pairs Shortest Paths"
- Works on a matrix of size $i * j$
- Runtime Complexity: $O(V^3)$
- When to use?
  - Calculating shortest distances between EVERY pair of nodes

- Dijkstra's algorithm is very popular SPF algorithm
- Best use case is from one source to all destination nodes

Algorithm	Runtime Complexities
A*	$O(E)$
Dijkstra's	$O(E \log (V))$
Bellman-Ford's	$O(VE)$
Floyd-Warshall's	$O(V^3)$

Proof of Correctness

How can we be sure that Dijkstra's algorithm provides us the shortest possible path between two nodes? Let's go through the following explanation to understand whether this algorithm always gives us the shortest possible path.
Consider the following notations:

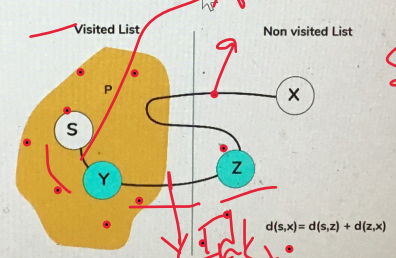
```
D(s,u) = the minimum distance as calculated by Dijkstra's algorithm between nodes s and u
d(s,u) = the actual minimum distance between nodes s and u
```

According to Dijkstra's Algorithm, `D(s,u) = d(s,u)`

Proof:

- Let us start by assuming that Dijkstra's Algorithm is **incorrect**.
  - This means there would be some vertices left when a vertex `u` is included into the Visited List which indicates => `D(s,u) > d(s,u)`
- Let `x` be the first of these vertices that was pushed into the Visited List. Then,
  - When node `x` is included into the Visited List: `D(s,x) > d(s,x)`
  - This implies that all previous vertices, say `z` that were included into the Visited List signifies `D(s,z) = d(s,z)`
  - Let's look at the moment when this node `x` is included into the Visited List.
    - Let `P` be the (real) shortest path from

- Let `P` be the (real) shortest path from `s` to `x`
- Let `z` be the first node that is not in the "Visited List" and is along the shortest path.
- Let `y` be the predecessor node of `z` on the shortest path `P`
- Have a look at the diagram below for better understanding:



We can conclude the following:

`D(s,y) = d(s,y)` ... **(1)** (This means the minimum distance `s` to `y` computed by the algorithm = actual min. distance `s` to `y` as `y` is included before `x`)
`D(s,z) = D(s,y) + edge_cost(y,z) = d(s,y) + edge_cost(y,z)` ... **(2)**
`D(s,x) ≤ D(s,z)` ...**(3)** (Because the next vertex included by the algorithm is `x`)

- We can now finally deduce that:

```
D(s,x) ≤ D(s,z)                                  ... From (3)
       = d(s,y) + edge_cost(y,z)                 ... From (2)

       ≤ d(s,y) + edge_cost(y,z) + d(z,x)

       = d(s,x)
```

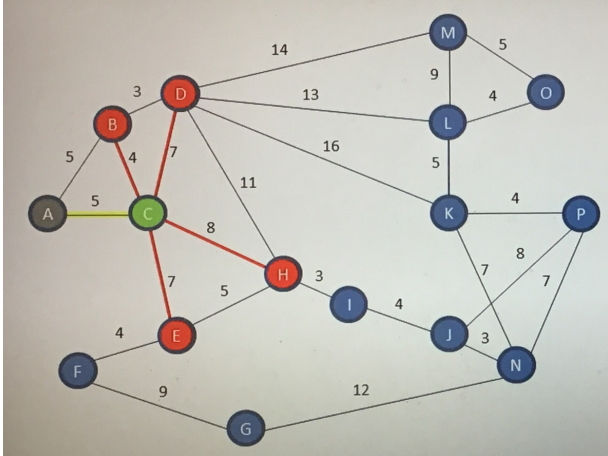
- The above result **contradicts** the assumed fact of Dijkstra's algorithm being incorrect earlier.
- Hence, by **proof of contradiction**, we can say that **Dijkstra's algorithm always gives us the shortest possible path** between 2 nodes which is: `D(s,x) should be equal to d(s,x)`

A Star Search.

Graph Data Structure 6. The A* Pathfinding Algorithm

Open B C D H E

Closed A

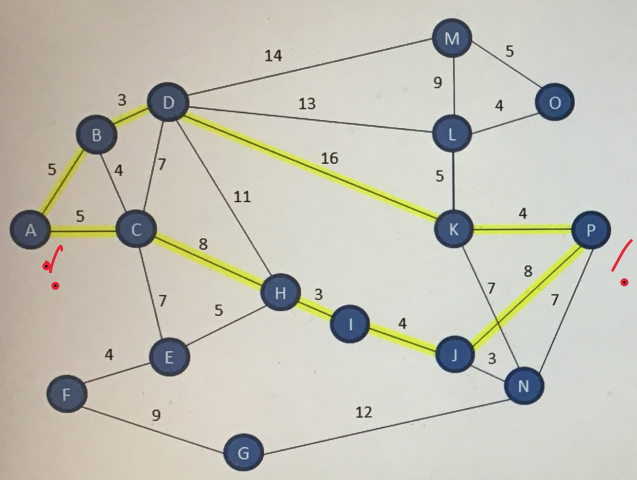


Vertex	Distance from A (g)	Heuristic distance (h)	f = g + h	Previous vertex
A	0	16	16	
B	5 9	17	22 26	A C
C	5	13	18	A
D	12	16	28	C
E	12	16	28	C
F		20		
G		17		
H	13	11	24	C
I		10		
J		8		
K		4		
L		7		
M		10		
N		7		
O		5		

Exit full screen (

Open E M L K J P N

Closed A C B H D I

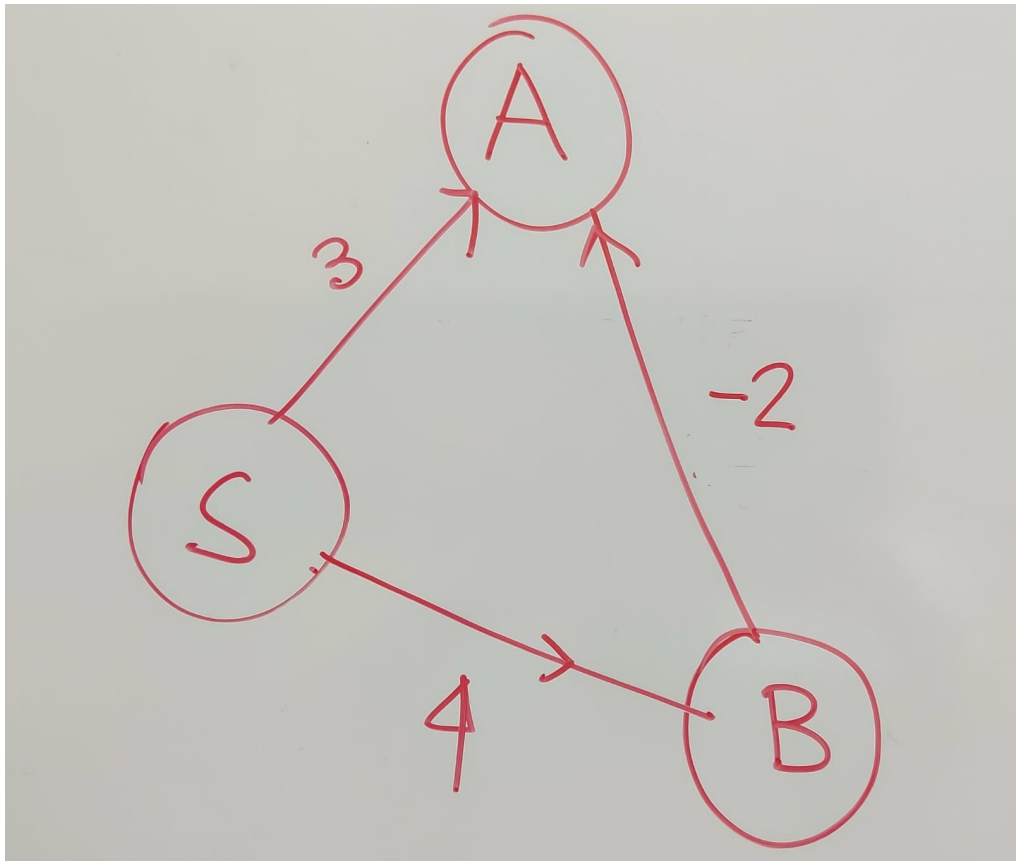


Vertex	Distance from A (g)	Heuristic distance (h)	f = g + h	Previous vertex
A	0	16	16	
B	5	17	22	A
C	5	13	18	A
D	8	16	24	B
E	12	16	28	C
F		20		
G		17		
H	13	11	24	C
I	16	10	26	H
J	20	8	28	I
K	24	4	28	D
L	21	7	28	D
M	22	10	32	D
N	23	7	30	J
O		5		
P	28	0	28	J

Bellman Ford Algorithm

Debarka Sengupta

Negative edge – Dijkstra is not guaranteed to work



Relaxation Step

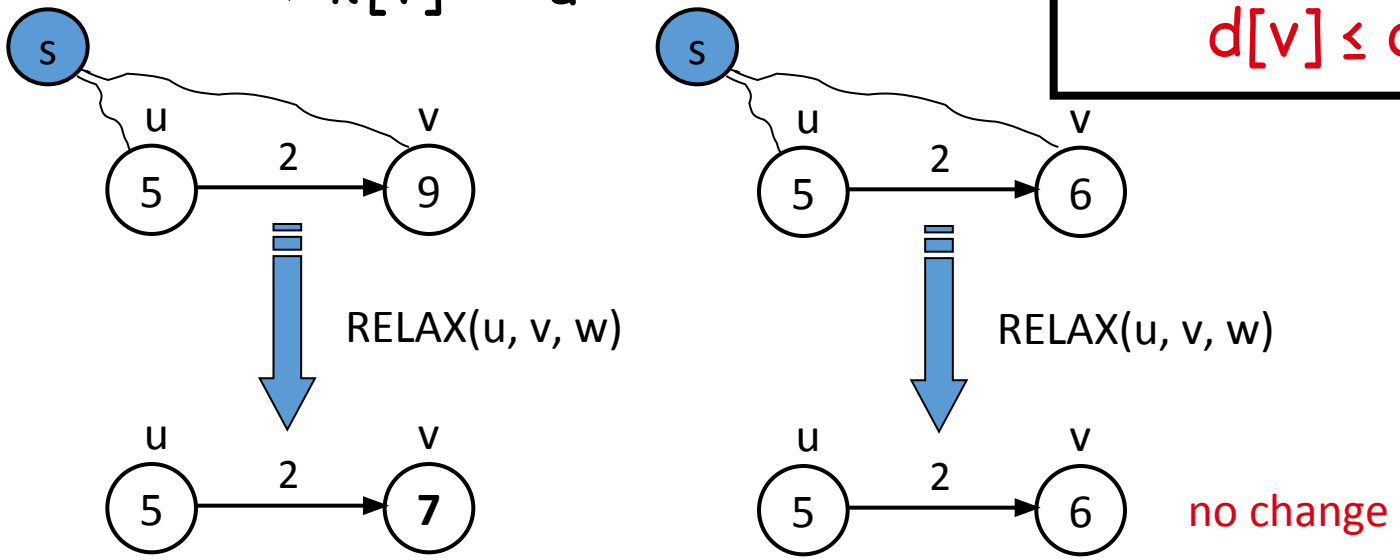
- **Relaxing** an edge (u, v) = testing whether we can improve the shortest path to v found so far by going through u

  If $d[v] > d[u] + w(u, v)$

  we can improve the shortest path to v

  $\Rightarrow d[v]=d[u]+w(u,v)$

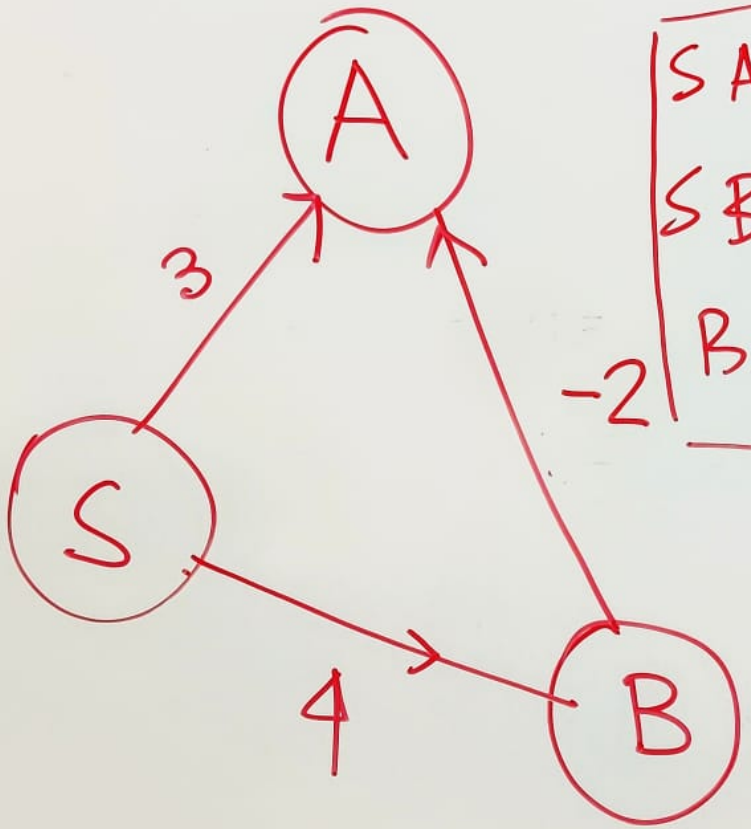
  $\Rightarrow \pi[v] \leftarrow u$



After relaxation:

$d[v] \leq d[u] + w(u, v)$

Crude description of BF algo



Edges.

SA	(3)
SB	(4)
BA	(-2)

-2

S	A	B
0	∞	∞
0	3	∞
0	3	4
0	-2	4
0	-2	4

it 1

∴ No change.

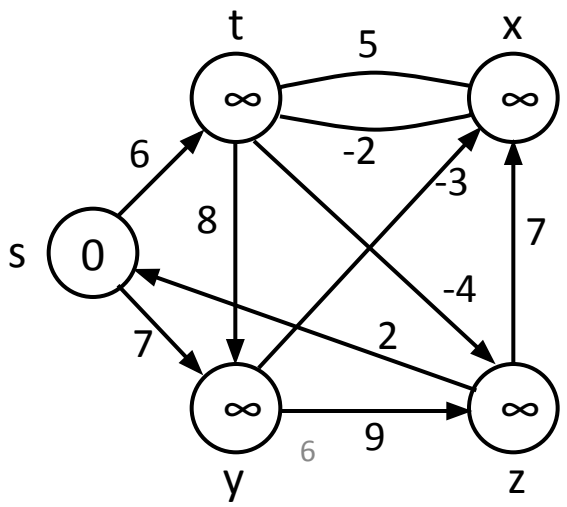
Bellman-Ford Algorithm

- Single-source shortest path problem
  - Computes $\delta(s, v)$ and $\pi[v]$ for all $v \in V$
- Allows negative edge weights - can detect negative cycles.
  - Returns TRUE if no negative-weight cycles are reachable from the source s
  - Returns FALSE otherwise $\Rightarrow$ no solution exists

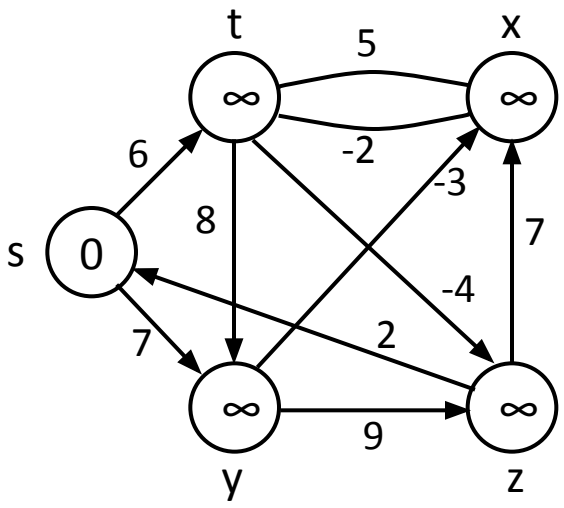
Bellman-Ford Algorithm (cont'd)

- Idea:
  - Each edge is relaxed |V–1| times by making |V-1| passes over the whole edge set.
  - To make sure that each edge is relaxed exactly |V – 1| times, it puts the edges in an unordered list and goes over the list |V – 1| times.

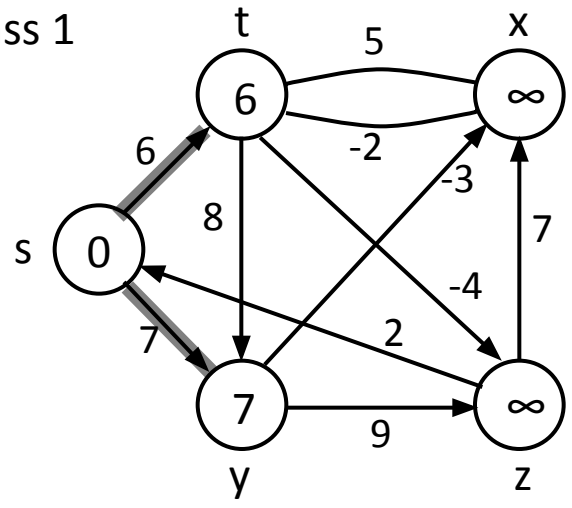
(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)



BELLMAN-FORD(V, E, w, s)



Pass 1

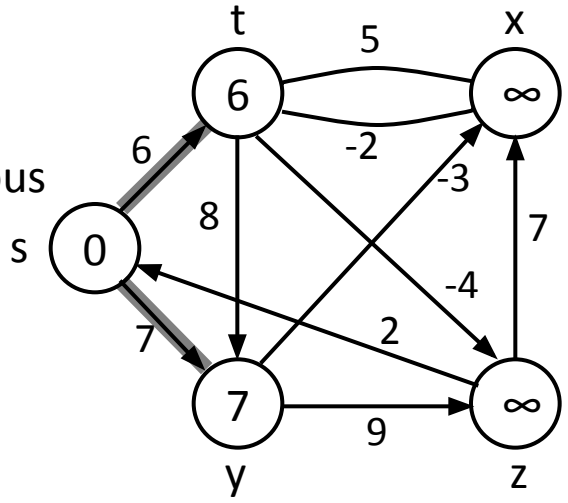


E: (t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

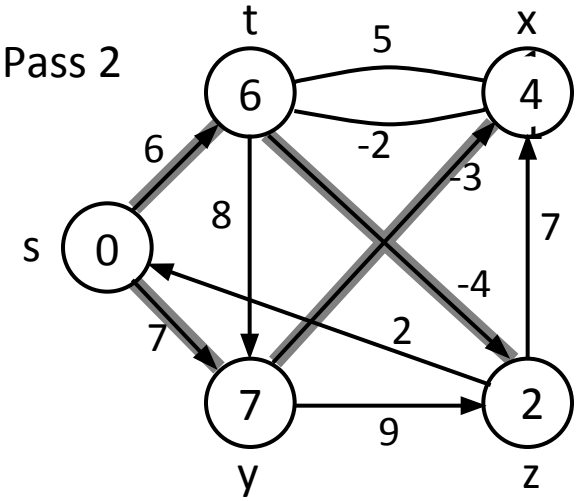
(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)

Example

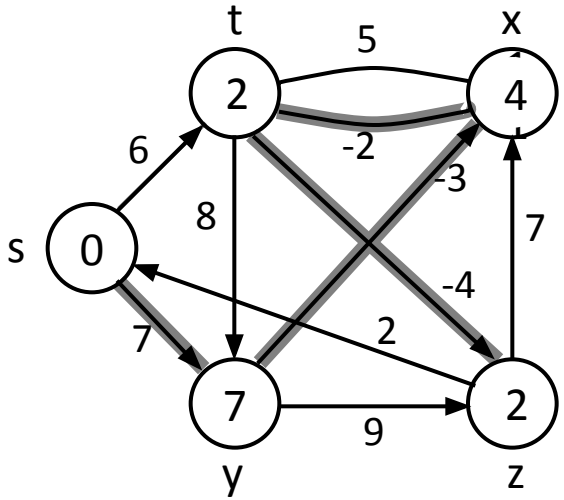
Pass 1 (from previous slide)



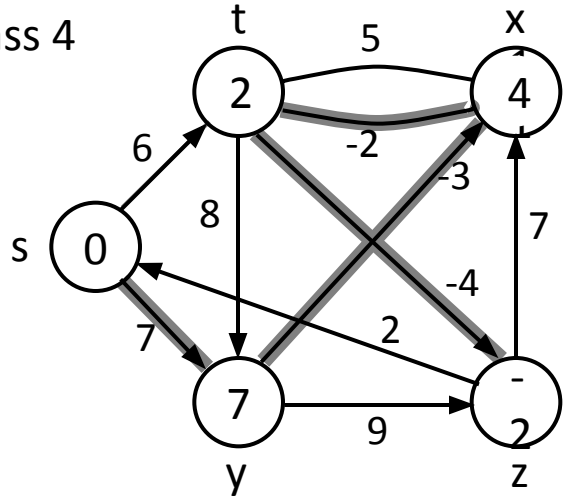
Pass 2



Pass 3

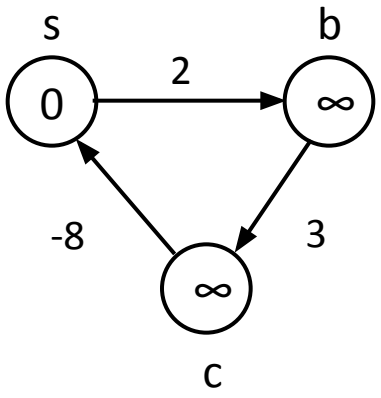


Pass 4

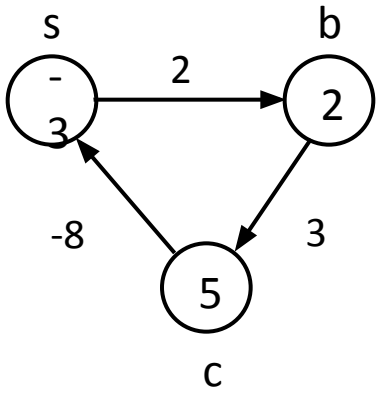


Detecting Negative Cycles (perform extra test after V-1 iterations)

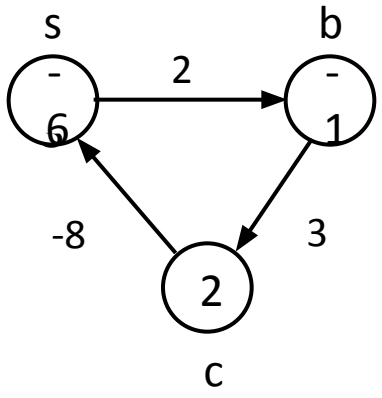
- **for** each edge (u, v) $\in$ E
- **do if** d[v] > d[u] + w(u, v)
- **then return** FALSE
- **return** TRUE



1st pass



2nd pass



(s,b) (b,c) (c,s)

Look at edge (s, b):

d[b] = -1
d[s] + w(s, b) = -4

$\Rightarrow$ d[b] > d[s] + w(s, b)

BELLMAN-FORD(V, E, w, s)

```
INITIALIZE-SINGLE-SOURCE(V, s)          ← Θ(V)
for i ← 1 to |V| - 1                    ← O(V)
    do for each edge (u, v) ∈ E         ← O(E)
            do RELAX(u, v, w)
for each edge (u, v) ∈ E                ← O(E)
    do if d[v] > d[u] + w(u, v)
            then return FALSE
return TRUE
```

O(VE)

Running time: O(V+VE+E)=O(VE)

Bellman-Ford detects negative cycles, i.e. if there is a negative cycle reachable from the source s, then for some edge

$$(u, v),\ d_{n-1}(v) > d_{n-1}(u) + w(u, v).$$

Suppose $v_0 \to v_1 \to \cdots \to v_k$ is a negative cycle reachable from s, where $v_0 = v_k$. Formally, this means we have

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

We proceed by contradiction. Suppose that we have $d_{n-1}(v_i) \leq d_{n-1}(v_{i-1}) + w(v_{i-1}, v_i)$ for all $i = 1, \ldots, k$. Then summing up the inequality for each vertex on the cycle, we get

$$\sum_{i=1}^{k} d_{n-1}(v_i) \leq \sum_{i=1}^{k} d_{n-1}(v_{i-1}) + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

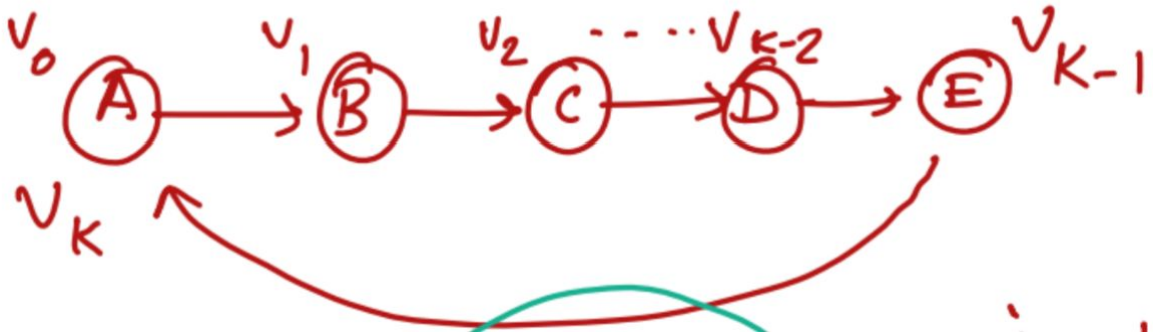
Observe that the first two terms are the same. That's because $v_0 = v_k$, so

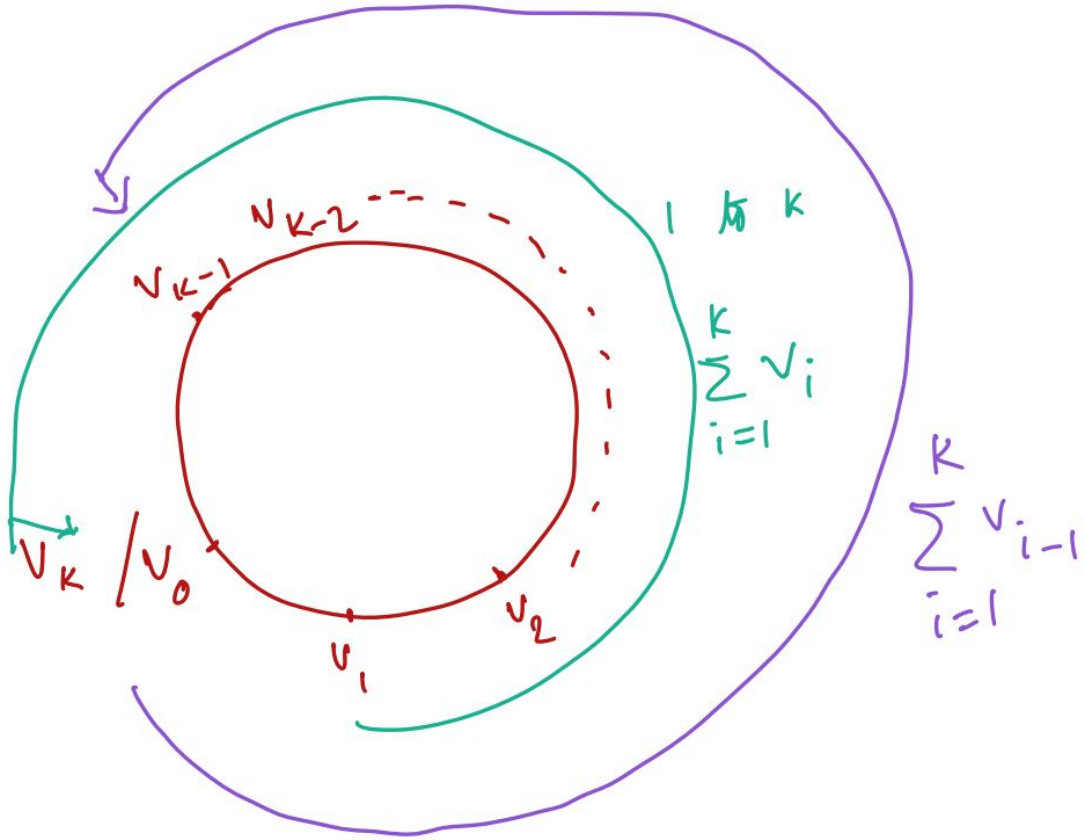
$$\sum_{i=1}^{k} d_{n-1}(v_{i-1}) = \sum_{i=0}^{k-1} d_{n-1}(v_i) = d_{n-1}(v_0) + \sum_{i=1}^{k-1} d_{n-1}(v_i) = \sum_{i=1}^{k} d_{n-1}(v_i)$$

Because those two terms are the same, we can cancel them out to get

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$$

contradicting the supposition that $v_0 \to v_1 \to \cdots \to v_k$ is a negative cycle.


v_0
v_1
v_2
v_{k-2}
v_{k-1}
v_k
A
B
C
D
E


v_{k-2}
v_{k-1}
v_k / v_0
v_1
v_2
1 加 k
$\sum_{i=1}^{k} v_i$
$\sum_{i=1}^{k} v_{i-1}$

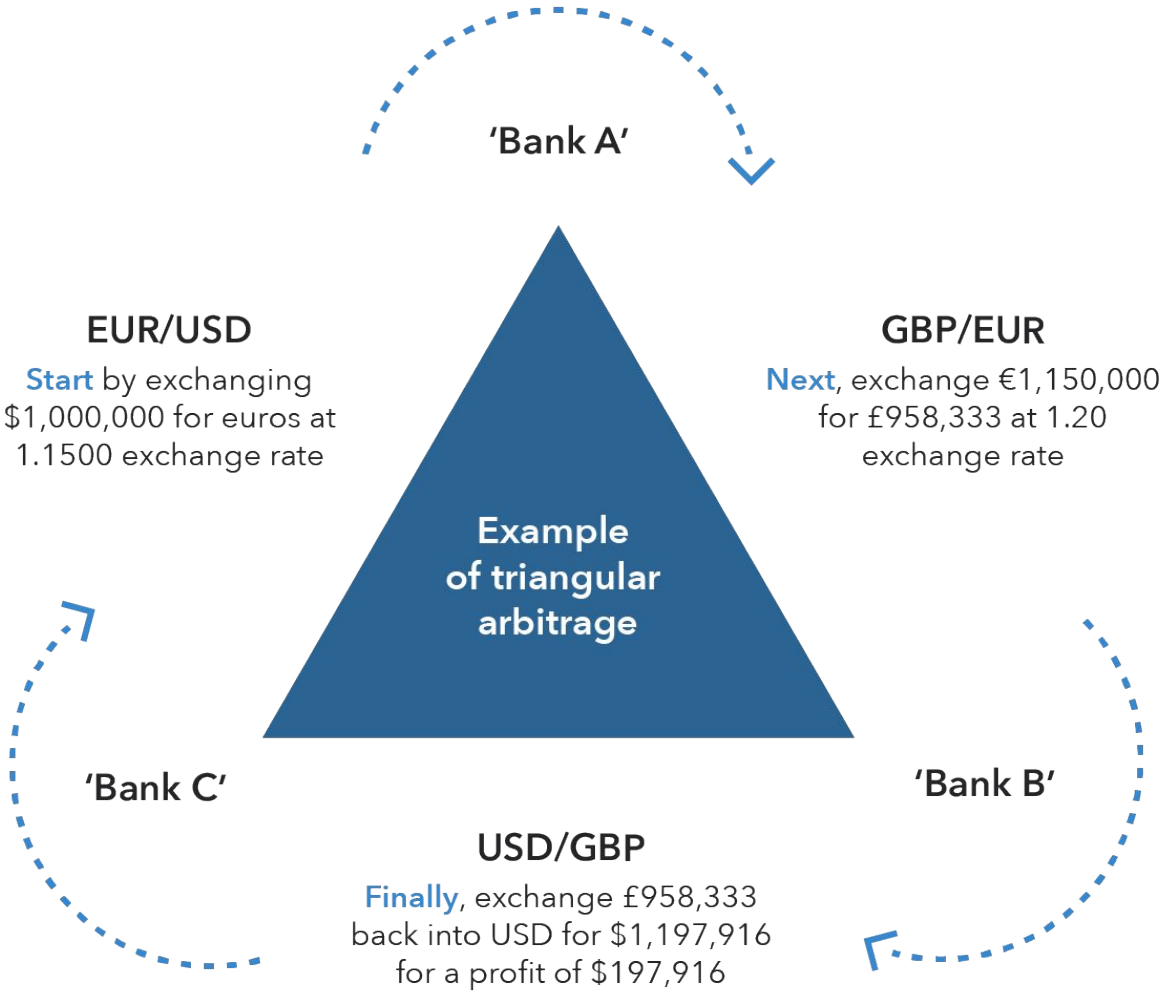
If the graph has no negative cycles, then the distance estimates on the last iteration are equal to the true shortest distances.

That is,

$$d_{n-1}(v) = \delta(s, v)$$

for all vertices v.

Self read: https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture14.pdf



ARBITRAGE

Here you have the solution: https://courses.csail.mit.edu/6.046/spring04/handouts/ps7sol.pdf

Network Flow I
(ADA 2023, CSE 222)

Diptapriyo Majumdar

April 3, 2023

Outline

1. Network Flow

2. Max-Flow Min-Cut Theorem

Capacity c $\quad c: E(G) \to \mathbb{R}^+$

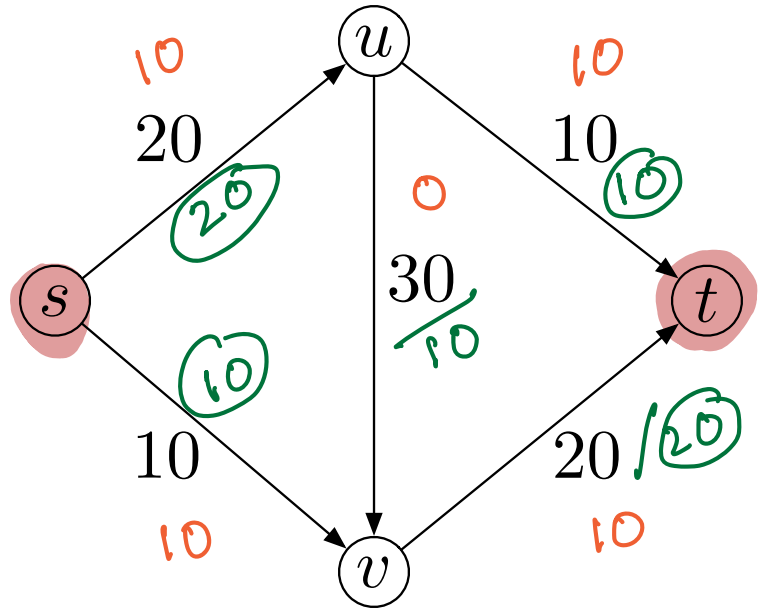
(s,u) - 20

(s,v) - 10

(u,v) - 30

(u,t) - 10

(v,t) - 20



flow f

$f: E(G) \to \mathbb{R}^+$

(s,u) - 10/20

(u,t) - 10/10

(s,v) - 10/10

(v,t) - 10/20

(u,v) - 0/10

Capacity Condition: $f(e) \le c(e)$

Flow Conservation:

$$\sum_{e \text{ into } u} f(e) = \sum_{e \text{ out of } u} f(e) \quad \text{for every } u \text{ except } s \text{ and } t.$$

- **Flow Network:** A directed graph $G = (V, E)$ with $s, t \in V(G)$ (s is source and t is sink). Every edge $e \in E(G)$ has a nonnegative *capacity* $\mathrm{c} : E(G) \to \mathbb{Z}^+$.
- **Defining flow:** What is *flow* or traffic in a network? An s-t *flow* in a network is a function $\mathrm{f} : E(G) \to \mathbb{Z}^+$ satisfying the following two conditions.
  - **Capacity condition:** For every $e \in E(G)$, $\mathrm{f}(e) \leq \mathrm{c}(e)$.
  - **Flow conservation condition:** For each u other than s and t, we have
  $$\sum_{v:(v,u)\in E(G)} \mathrm{f}(v,u) = \sum_{v:(u,v)\in E(G)} \mathrm{f}(u,v)$$
  More specifically $\mathrm{f}^{\mathrm{out}}(u) = \sum_{v:(u,v)\in E(G)} \mathrm{f}(u,v) = \sum_{e \text{ out of } u} \mathrm{f}(e)$ and $\mathrm{f}^{\mathrm{in}}(u) = \sum_{v:(v,u)\in E(G)} \mathrm{f}(v,u) = \sum_{e \text{ into } u} \mathrm{f}(e)$.
- **Input:** A flow network with capacity conditions flow conservation conditions.

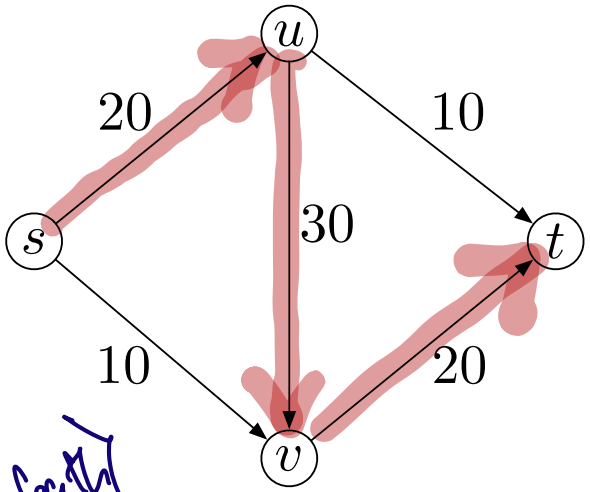
(s,t) flow

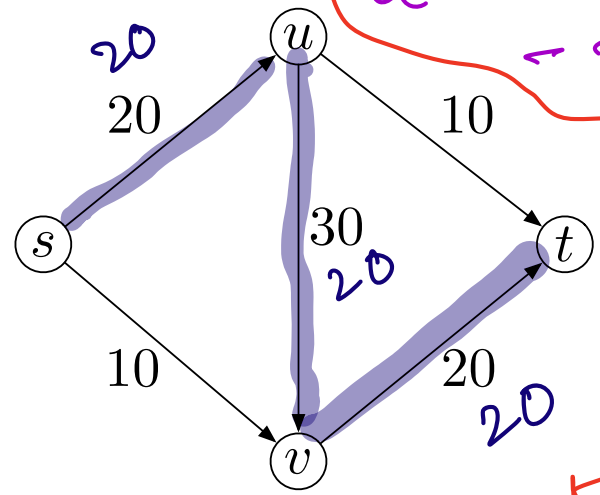
- **Output:** A flow $\mathrm{f} : E(G) \to \mathbb{Z}^+$ that satisfies capacity conditions and flow conservation conditions, and maximize $\sum_{e \text{ out of } s} \mathrm{f}(e) = \sum_{u:(s,u)\in E(G)} \mathrm{f}(s,u)$.

$s \to u \to v \to t$

Bottleneck capacity of an (s,t)-path

$\min\{c(e) \mid e \in (s,t)\text{-path}\}$



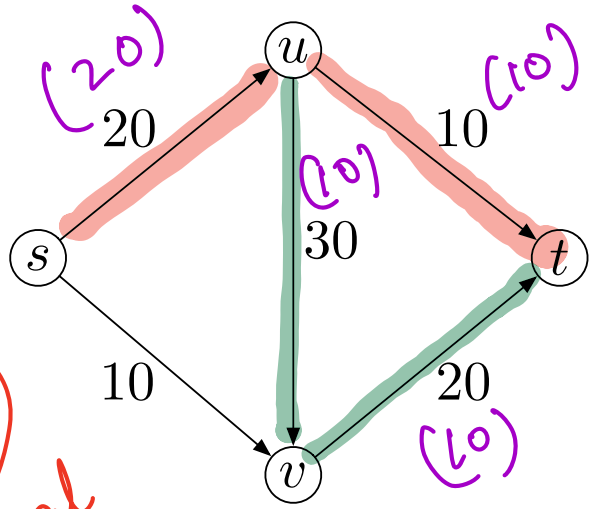


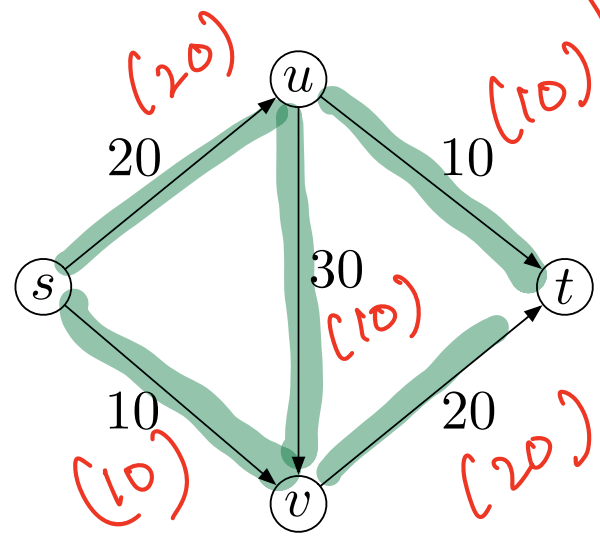
$c(u,v) - f(u,v) = 30 - 10 = 20$

residual capacity

If $f(u,v) = c(u,v)$

then (u,v) is saturated.

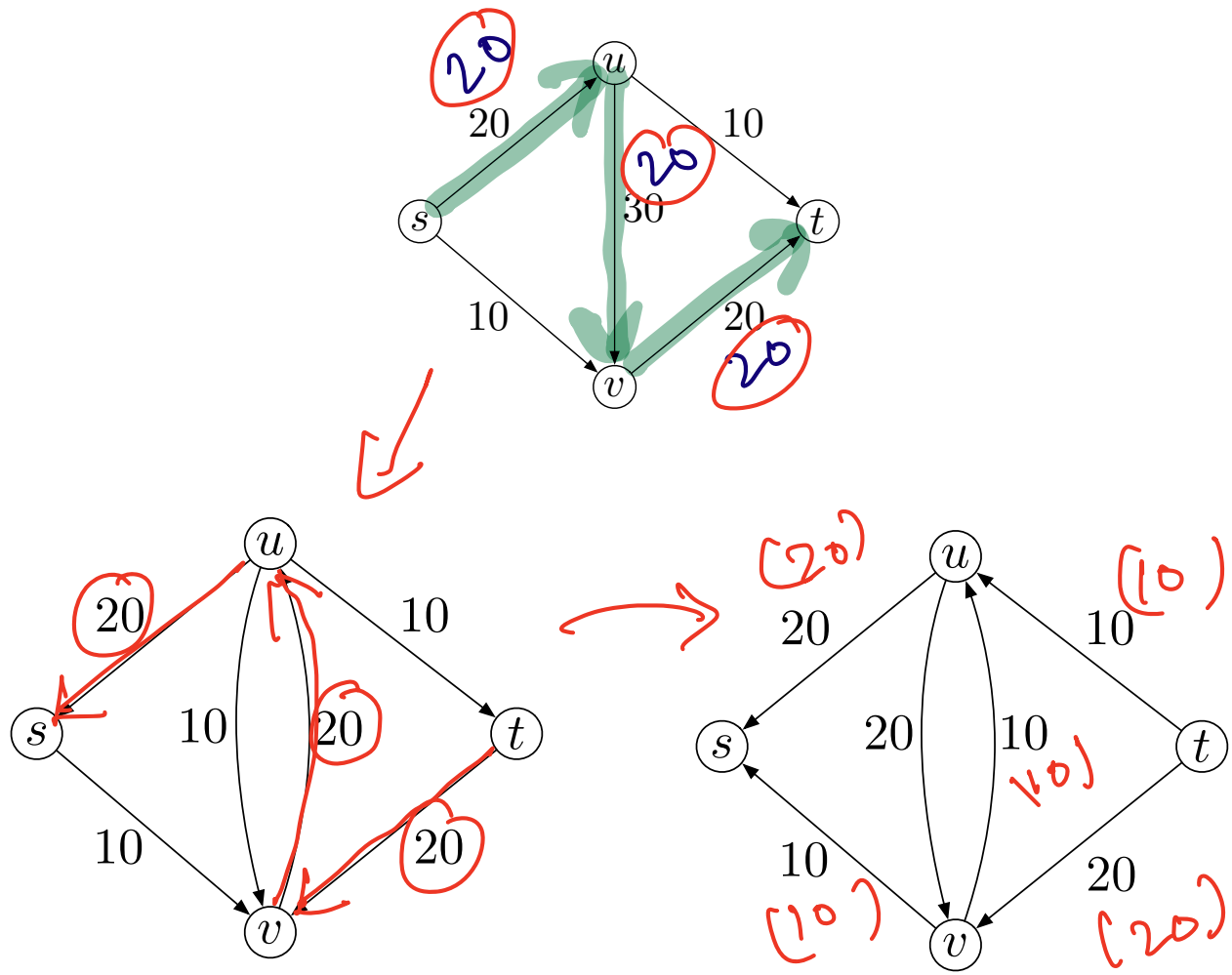




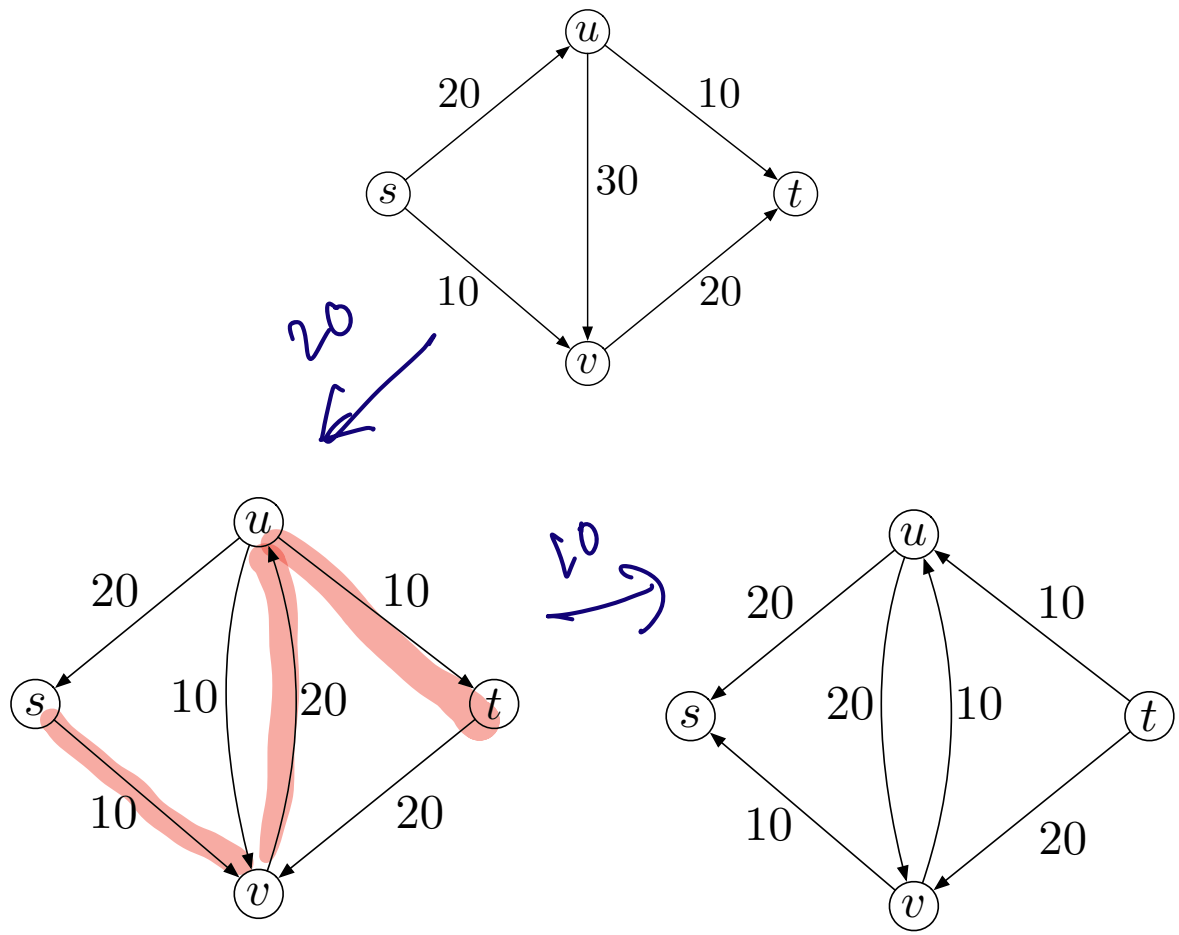
$c(v,t) - f(v,t) = 20 - 10 = 10$

residual capacity

- **Residual Graph:** Given a flow network G and a flow f on G, the *residual graph* G_{f} of G with respect to f as follows.



- **Augmenting Path:** An s-t path in residual graph is called *augmenting path*.

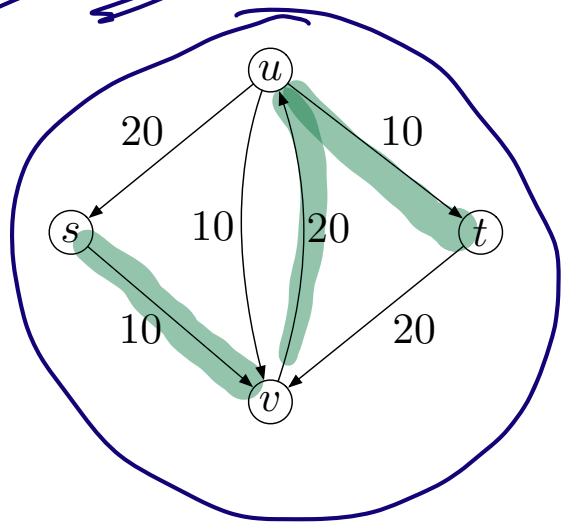


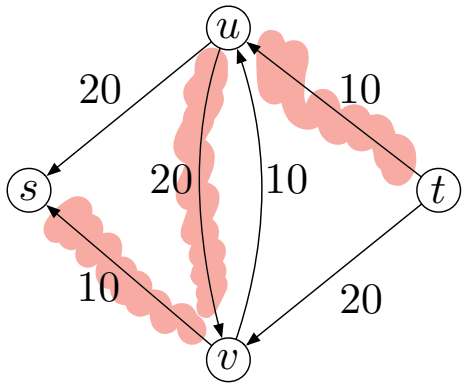
- Let G be a flow network and a f be a flow in G. Then, G_{f} has same vertex set as G.

$c(e) - f(e)$

- For every edge e, if $0 < \mathrm{f}(e) < \mathrm{c}(e)$, then $\cancel{\mathrm{f}(e) - \mathrm{c}(e)}$ is called *residual capacity*. By this forward edge e, this residual amount of flow $\mathrm{c}(e) - \mathrm{f}(e)$ can still be sent.

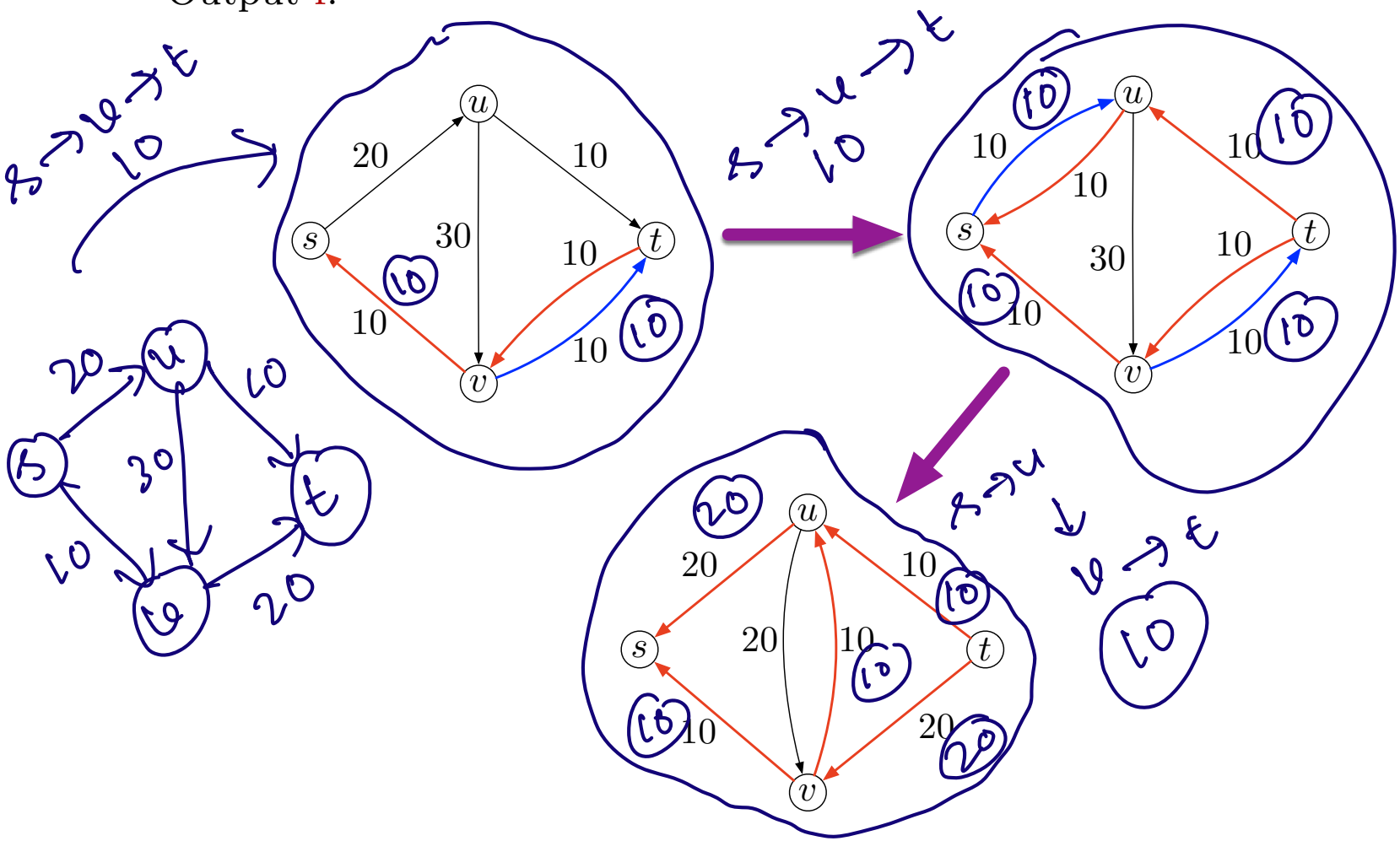
- For every edge e, if $0 < \mathrm{f}(e) < \mathrm{c}(e)$, then a backward edge e^{rev} is created with $\mathrm{c}(e^{rev}) = \mathrm{f}(e)$ in G_{f}.

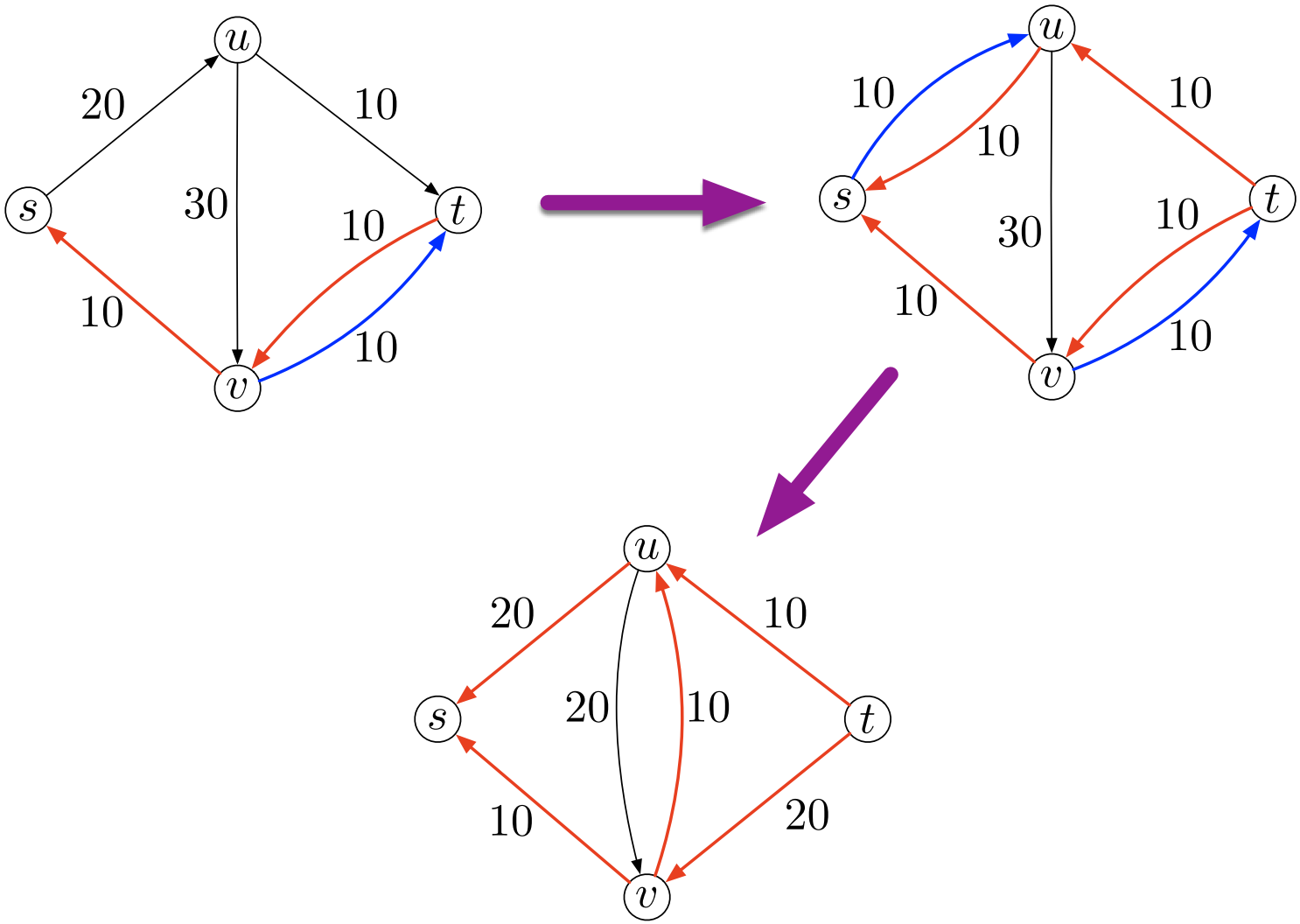




- For all $e \in E(G)$, initialize $\mathrm{f}(e) = 0$ and $G_{\mathrm{f}} = G$.
- As long as there is an augmenting path P in G_{f}
  - Send the maximum amount of flow via the path P from s to t.
  - Compute G_{f}.
- Output f.

bottleneck capacity

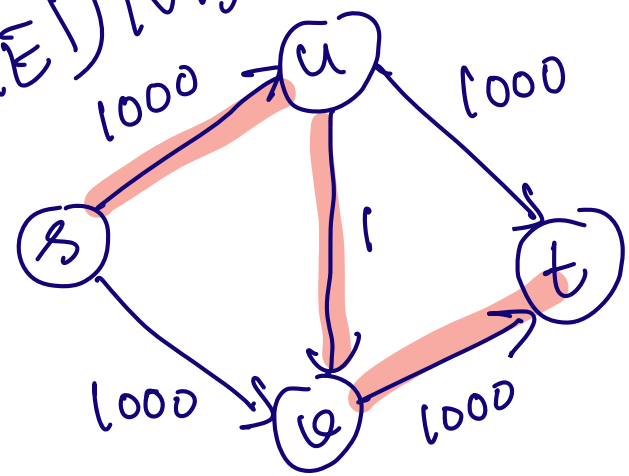


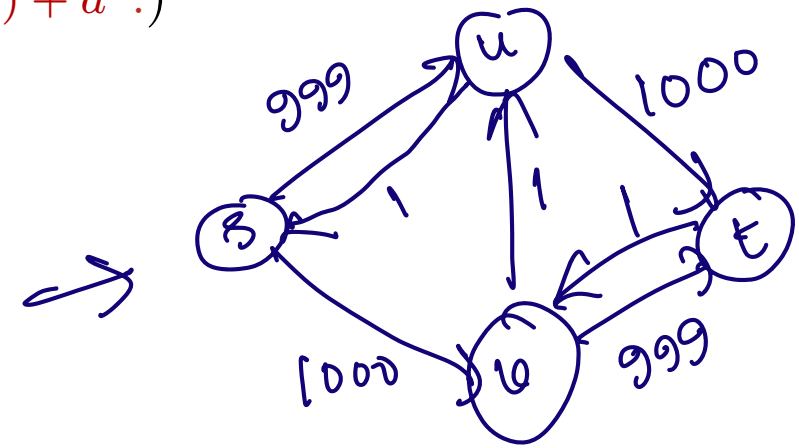

u
s
t
v
20
10
30
10
10
10
u
s
t
v
10
10
10
30
10
10
10
u
s
t
v
20
10
20
10
10
20

Running time

- For all $e \in E(G)$, initialize $\mathrm{f}(e) = 0$ and $G_{\mathrm{f}} = G$.
- As long as there is an augmenting path P in G_{f}
  - Send the maximum amount of flow via the path P from s to t.
    (Find the minimum capacity a^* across all edges of P. Add a to all the edges of P, i.e. for all $e \in P$, $\mathrm{f}(e) \leftarrow \mathrm{f}(e) + a^*$.)
  - Compute G_{f}.
- Output f.

$O(\text{flow value} * (|V| + |E|)|V|)$

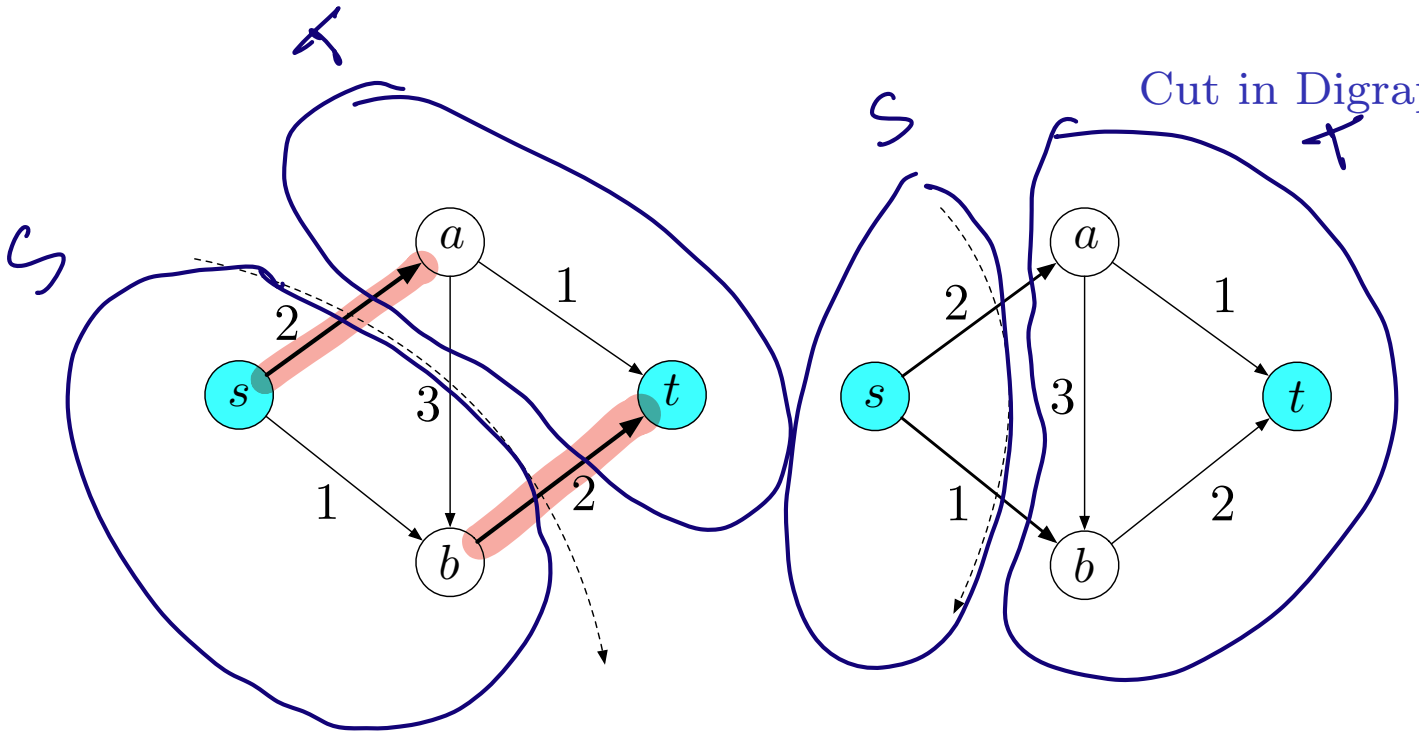




Outline

1. Network Flow
2. Max-Flow Min-Cut Theorem

Cut in Digraphs

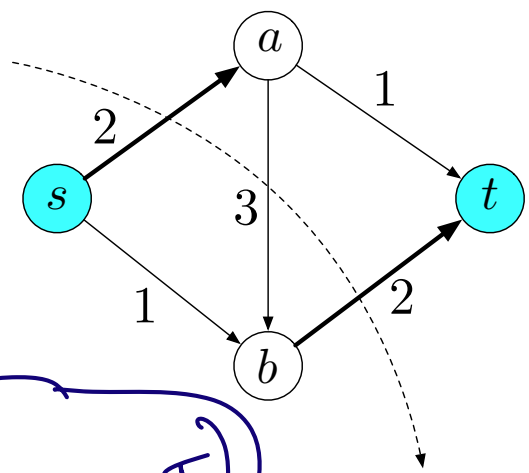


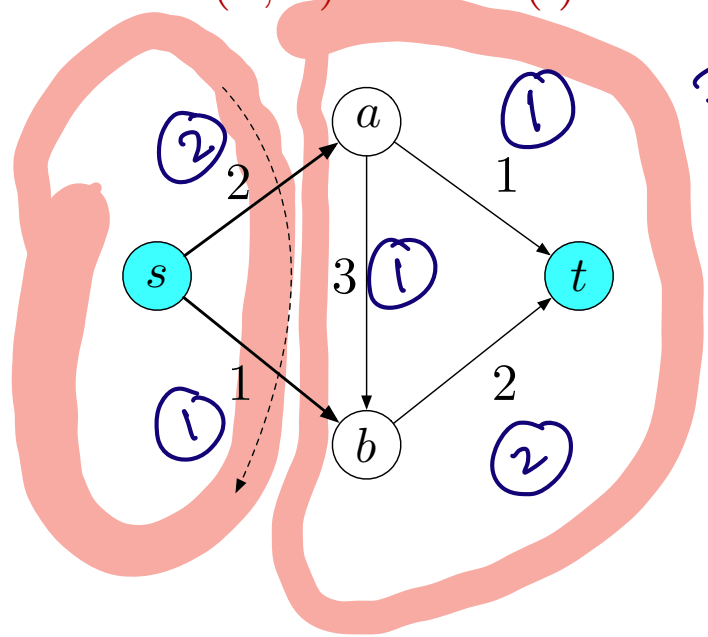
- In a flow network (G, s, t, c), an (s,t)-cut is a nonempty $S \subset V(G)$ such that $s \in S$, and $t \in V \setminus S$.
- Capacity of an (s,t)-cut (S,T) is $\sum_{u \in S} \sum_{v \in T} \mathrm{c}(u,v)$. $c(S,T)$
- Capacity of $(\{s,b\},\{a,t\})$ is 4 and capacity of $(\{s\},\{a,b,t\})$ is 3.

Max-Flow Min-Cut Theorem

Let (G, s, t, c) be a flow network and $\mathrm{f} : E(G) \to \mathbb{Z}^+$ be a valid (s, t)-flow. Then, the followings are equivalent.

1. $\mathrm{Value(f)}$ is a maximized.
2. G_{f} has no augmenting paths.
3. There exists an (s, t)-cut (S, T) such that $\mathrm{c}(S, T) = \mathrm{Value(f)}$.





flow

$f(s,a) = 2$

$f(s,b) = 1$

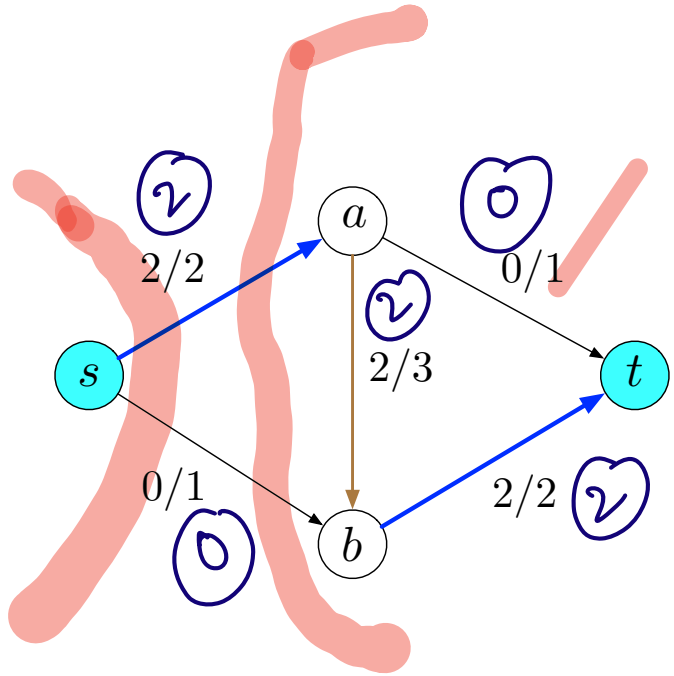
$f(b,t) = 2$

$f(a,t) = 1$

$f(a,b) = 1$

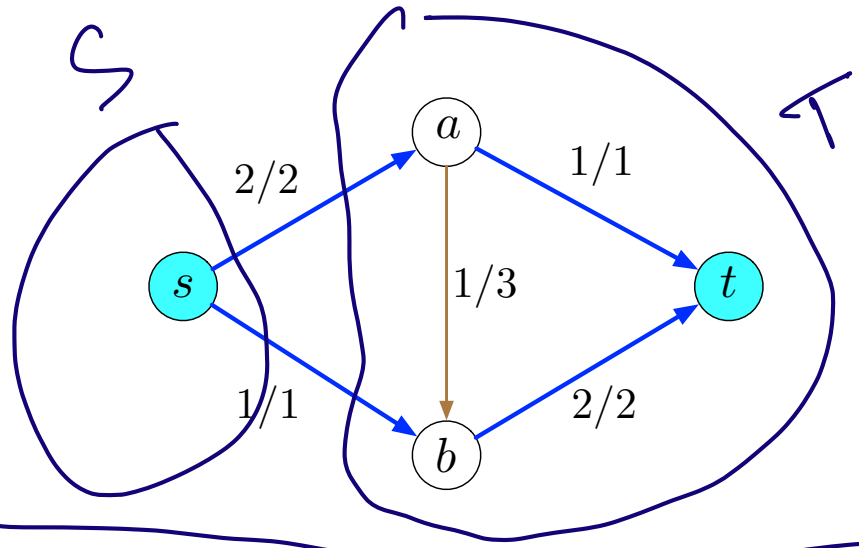
value(f) $\leq$ c(S,T)

Flow across a cut



- $\mathrm{f}(S,T) = \sum_{u \in S} \sum_{v \in T} \mathrm{f}(u,v) - \sum_{u \in S} \sum_{v \in T} \mathrm{f}(v,u)$ [flow across (S,T)-cut].

Flow across a cut



- $\mathrm{f}(S,T) = \sum_{u \in S} \sum_{v \in T} \mathrm{f}(u,v) - \sum_{v \in T} \sum_{u \in S} \mathrm{f}(v,u)$ [flow across the cut (S,T)].
- Flow across $(\{s\},\{a,b,t\})$ is 3.
- Flow across $(\{s,a\},\{b,t\})$ is also 3.

References

- Algorithm Design Chapter 7 by Kleinberg and Tardos.
- Lecture notes by Jeff Erickson.

Max flow / Min Cut

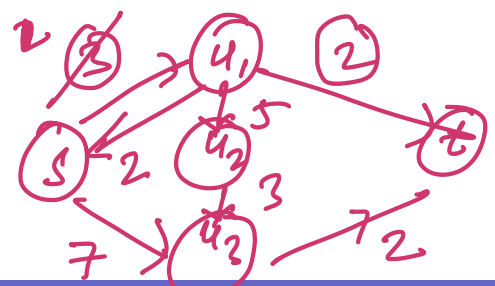
1. How determine the min-cut.

Applications of Network Flow

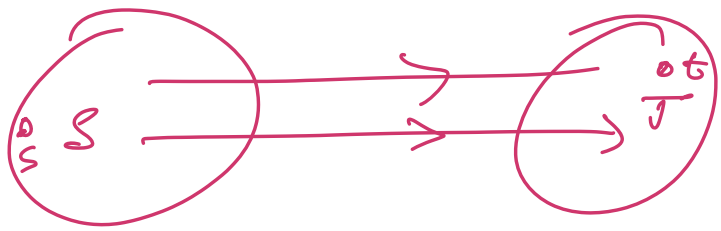
T. M. Murali

November 16, 18, 2009

1. Discover augmenting path. (BFS/DFS).
2.



1. obtain the residual graph G'
2. Find out nodes that are reachable from S in G' $\rightarrow$ S $\rightarrow$ set
3. Rest T



Maximum Flow and Minimum Cut

- Two rich algorithmic problems.
- Fundamental problems in combinatorial optimization.
- Beautiful mathematical duality between flows and cuts.
- Numerous non-trivial applications:
  - Bipartite matching.
  - Data mining.
  - Project selection.
  - Airline scheduling.
  - Baseball elimination.
  - Image segmentation.
  - Network connectivity.
  - Open-pit mining.
  - Network reliability.
  - Distributed computing.
  - Egalitarian stable matching.
  - Security of statistical data.
  - Network intrusion detection.
  - Multi-camera scene reconstruction.
  - Gene function prediction.

Maximum Flow and Minimum Cut

- Two rich algorithmic problems.
- Fundamental problems in combinatorial optimization.
- Beautiful mathematical duality between flows and cuts.
- Numerous non-trivial applications:
  - Bipartite matching.
  - Data mining.
  - Project selection.
  - Airline scheduling.
  - Baseball elimination.
  - Image segmentation.
  - Network connectivity.
  - Open-pit mining.
  - Network reliability.
  - Distributed computing.
  - Egalitarian stable matching.
  - Security of statistical data.
  - Network intrusion detection.
  - Multi-camera scene reconstruction.
  - Gene function prediction.

Maximum Flow and Minimum Cut

- Two rich algorithmic problems.
- Fundamental problems in combinatorial optimization.
- Beautiful mathematical duality between flows and cuts.
- Numerous non-trivial applications:
  - Bipartite matching.
  - Data mining.
  - Project selection.
  - Airline scheduling.
  - Baseball elimination.
  - Image segmentation.
  - Network connectivity.
  - Open-pit mining.
  - Network reliability.
  - Distributed computing.
  - Egalitarian stable matching.
  - Security of statistical data.
  - Network intrusion detection.
  - Multi-camera scene reconstruction.
  - Gene function prediction.
- We will only sketch proofs. Read details from the textbook.

Matching in Bipartite Graphs

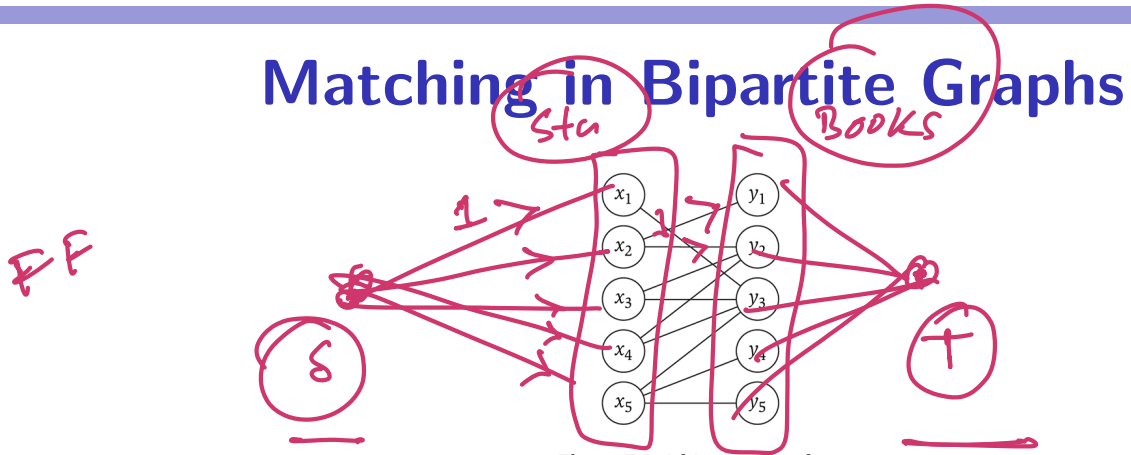


Figure 7.1 A bipartite graph.

- *Bipartite Graph*: a graph $G(V, E)$ where
  1. $V = X \cup Y$, X and Y are disjoint and
  2. $E \subseteq X \times Y$.
- Bipartite graphs model situations in which objects are matched with or assigned to other objects: e.g., marriages, residents/hospitals, jobs/machines.

Matching in Bipartite Graphs

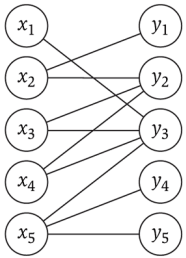


Figure 7.1 A bipartite graph.

- *Bipartite Graph*: a graph $G(V, E)$ where
  1. $V = X \cup Y$, X and Y are disjoint and
  2. $E \subseteq X \times Y$.
- Bipartite graphs model situations in which objects are matched with or assigned to other objects: e.g., marriages, residents/hospitals, jobs/machines.
- A *matching* in a bipartite graph G is a set $M \subseteq E$ of edges such that each node of V is incident on at most edge of M.
- A set of edges M is a *perfect matching* if every node in V is incident on exactly one edge in M.

Bipartite Graph Matching Problem

Bipartite Matching

INSTANCE: A Bipartite graph G.

SOLUTION: The matching of largest size in G.

Algorithm for Bipartite Graph Matching

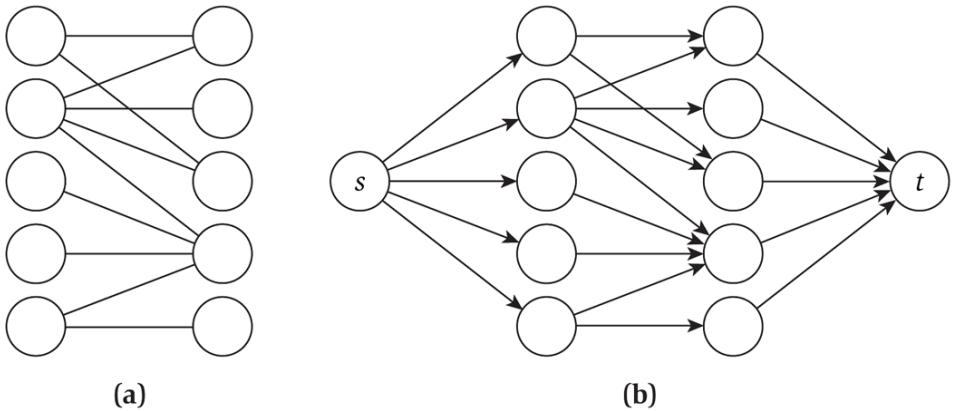


Figure 7.9 (a) A bipartite graph. (b) The corresponding flow network, with all capacities equal to 1.

- Convert G to a flow network G': direct edges from X to Y, add nodes s and t, connect s to each node in X, connect each node in Y to t, set all edge capacities to 1.
- Compute the maximum flow in G'.
- Claim: the value of the maximum flow is the size of the maximum matching.

Correctness of Bipartite Graph Matching Algorithm

- Matching $\rightarrow$ flow: if there is a matching with k edges in G, there is an s-t flow of value k in G'.

Correctness of Bipartite Graph Matching Algorithm

- Matching $\rightarrow$ flow: if there is a matching with k edges in G, there is an s-t flow of value k in G'.
- Flow $\rightarrow$ matching: if there is an integer-valued flow f' in G' with value k, there is a matching M in G with k edges.

Correctness of Bipartite Graph Matching Algorithm

- Matching $\rightarrow$ flow: if there is a matching with k edges in G, there is an s-t flow of value k in G'.
- Flow $\rightarrow$ matching: if there is an integer-valued flow f' in G' with value k, there is a matching M in G with k edges.
  - There is an integer-valued flow f of value $k \Rightarrow$ flow along any edge is 0 or 1.
  - Let M be the set of edges not incident on s or t with flow equal to 1.

Correctness of Bipartite Graph Matching Algorithm

- Matching $\rightarrow$ flow: if there is a matching with k edges in G, there is an s-t flow of value k in G'.
- Flow $\rightarrow$ matching: if there is an integer-valued flow f' in G' with value k, there is a matching M in G with k edges.
  - There is an integer-valued flow f of value $k \Rightarrow$ flow along any edge is 0 or 1.
  - Let M be the set of edges not incident on s or t with flow equal to 1.
  - Claim: M contains k edges.

Correctness of Bipartite Graph Matching Algorithm

- Matching $\rightarrow$ flow: if there is a matching with k edges in G, there is an s-t flow of value k in G'.
- Flow $\rightarrow$ matching: if there is an integer-valued flow f' in G' with value k, there is a matching M in G with k edges.
  - There is an integer-valued flow f of value $k \Rightarrow$ flow along any edge is 0 or 1.
  - Let M be the set of edges not incident on s or t with flow equal to 1.
  - Claim: M contains k edges.
  - Claim: Each node in X (respectively, Y) is the tail (respectively, head) of at most one edge in M.

Correctness of Bipartite Graph Matching Algorithm

- Matching $\rightarrow$ flow: if there is a matching with k edges in G, there is an s-t flow of value k in G'.
- Flow $\rightarrow$ matching: if there is an integer-valued flow f' in G' with value k, there is a matching M in G with k edges.
  - There is an integer-valued flow f of value $k \Rightarrow$ flow along any edge is 0 or 1.
  - Let M be the set of edges not incident on s or t with flow equal to 1.
  - Claim: M contains k edges.
  - Claim: Each node in X (respectively, Y) is the tail (respectively, head) of at most one edge in M.
- Conclusion: size of the maximum matching in G is equal to the value of the maximum flow in G'; the edges in this matching are those that carry flow from X to Y in G'.

Correctness of Bipartite Graph Matching Algorithm

- Matching $\rightarrow$ flow: if there is a matching with k edges in G, there is an s-t flow of value k in G'.
- Flow $\rightarrow$ matching: if there is an integer-valued flow f' in G' with value k, there is a matching M in G with k edges.
  - There is an integer-valued flow f of value $k \Rightarrow$ flow along any edge is 0 or 1.
  - Let M be the set of edges not incident on s or t with flow equal to 1.
  - Claim: M contains k edges.
  - Claim: Each node in X (respectively, Y) is the tail (respectively, head) of at most one edge in M.
- Conclusion: size of the maximum matching in G is equal to the value of the maximum flow in G'; the edges in this matching are those that carry flow from X to Y in G'.
- Read the book on what augmenting paths mean in this context.

Running time of Bipartite Graph Matching Algorithm

- Suppose G has m edges and n nodes in X and in Y.

Running time of Bipartite Graph Matching Algorithm

- Suppose G has m edges and n nodes in X and in Y.
- $C \leq n$.
- Ford-Fulkerson algorithm runs in $O(mn)$ time.

Bipartite Graphs without Perfect Matchings

- How do we determine if a bipartite graph G has a perfect matching?

Bipartite Graphs without Perfect Matchings

- How do we determine if a bipartite graph G has a perfect matching? Find the maximum matching and check if it is perfect.

Bipartite Graphs without Perfect Matchings

- How do we determine if a bipartite graph G has a perfect matching? Find the maximum matching and check if it is perfect.
- Suppose G has no perfect matching. Can we exhibit a short "certificate" of that fact?
- What can such certificates look like?

Bipartite Graphs without Perfect Matchings

- How do we determine if a bipartite graph G has a perfect matching? Find the maximum matching and check if it is perfect.
- Suppose G has no perfect matching. Can we exhibit a short "certificate" of that fact?
- What can such certificates look like?
- G has no perfect matching iff

Bipartite Graphs without Perfect Matchings

- How do we determine if a bipartite graph G has a perfect matching? Find the maximum matching and check if it is perfect.
- Suppose G has no perfect matching. Can we exhibit a short "certificate" of that fact?
- What can such certificates look like?
- G has no perfect matching iff the maximum capacity of a cut in G' is less than n. Therefore, the cut is a certificate.

Bipartite Graphs without Perfect Matchings

- How do we determine if a bipartite graph G has a perfect matching? Find the maximum matching and check if it is perfect.
- Suppose G has no perfect matching. Can we exhibit a short "certificate" of that fact?
- What can such certificates look like?
- G has no perfect matching iff the maximum capacity of a cut in G' is less than n. Therefore, the cut is a certificate.
- But we would like the certificate in terms of G.

Bipartite Graphs without Perfect Matchings

- How do we determine if a bipartite graph G has a perfect matching? Find the maximum matching and check if it is perfect.
- Suppose G has no perfect matching. Can we exhibit a short "certificate" of that fact?
- What can such certificates look like?
- G has no perfect matching iff the maximum capacity of a cut in G' is less than n. Therefore, the cut is a certificate.
- But we would like the certificate in terms of G.
  - For example, two nodes in X with one incident edge each with the same neighbour in Y.

Bipartite Graphs without Perfect Matchings

- How do we determine if a bipartite graph G has a perfect matching? Find the maximum matching and check if it is perfect.
- Suppose G has no perfect matching. Can we exhibit a short "certificate" of that fact?
- What can such certificates look like?
- G has no perfect matching iff the maximum capacity of a cut in G' is less than n. Therefore, the cut is a certificate.
- But we would like the certificate in terms of G.
  - For example, two nodes in X with one incident edge each with the same neighbour in Y.
  - Generally, a subset $A \subseteq X$ with neighbours $\Gamma(A) \subseteq Y$, such that $|A| > |\Gamma(A)|$.

Bipartite Graphs without Perfect Matchings

- How do we determine if a bipartite graph G has a perfect matching? Find the maximum matching and check if it is perfect.
- Suppose G has no perfect matching. Can we exhibit a short "certificate" of that fact?
- What can such certificates look like?
- G has no perfect matching iff the maximum capacity of a cut in G' is less than n. Therefore, the cut is a certificate.
- But we would like the certificate in terms of G.
  - For example, two nodes in X with one incident edge each with the same neighbour in Y.
  - Generally, a subset $A \subseteq X$ with neighbours $\Gamma(A) \subseteq Y$, such that $|A| > |\Gamma(A)|$.
- Hall's Theorem: Let $G(X \cup Y, E)$ be a bipartite graph such that $|X| = |Y|$. Then G either has a perfect matching or there is a subset $A \subseteq X$ such that $|A| > |\Gamma(A)|$. A perfect matching or such a subset can be computed in $O(mn)$ time.

Bipartite Graphs without Perfect Matchings

- How do we determine if a bipartite graph G has a perfect matching? Find the maximum matching and check if it is perfect.
- Suppose G has no perfect matching. Can we exhibit a short "certificate" of that fact?
- What can such certificates look like?
- G has no perfect matching iff the maximum capacity of a cut in G' is less than n. Therefore, the cut is a certificate.
- But we would like the certificate in terms of G.
  - For example, two nodes in X with one incident edge each with the same neighbour in Y.
  - Generally, a subset $A \subseteq X$ with neighbours $\Gamma(A) \subseteq Y$, such that $|A| > |\Gamma(A)|$.
- Hall's Theorem: Let $G(X \cup Y, E)$ be a bipartite graph such that $|X| = |Y|$. Then G either has a perfect matching or there is a subset $A \subseteq X$ such that $|A| > |\Gamma(A)|$. A perfect matching or such a subset can be computed in $O(mn)$ time. Read proof in the textbook.

Edge-Disjoint Paths

- A set of paths in a graph G is *edge disjoint* if each edge in G appears in at most one path.

Edge-Disjoint Paths

- A set of paths in a graph G is *edge disjoint* if each edge in G appears in at most one path.

Directed Edge-Disjoint Paths

INSTANCE: Directed graph $G(V, E)$ with two distinguished nodes s and t.

SOLUTION: The maximum number of edge-disjoint paths between s and t.

Mapping to the Max-Flow Problem

- Convert G into a flow network: s is the source, t is the sink, each edge has capacity 1.

Mapping to the Max-Flow Problem

- Convert G into a flow network: s is the source, t is the sink, each edge has capacity 1.
- Paths $\rightarrow$ flow: if there are k edge-disjoint paths from s to t, send one unit of flow along each to yield a flow with value k.
- Flow $\rightarrow$ paths: Suppose there is an integer-valued flow of value k. Are there k edge-disjoint paths? If so, what are they?

Mapping to the Max-Flow Problem

- Convert G into a flow network: s is the source, t is the sink, each edge has capacity 1.
- Paths $\rightarrow$ flow: if there are k edge-disjoint paths from s to t, send one unit of flow along each to yield a flow with value k.
- Flow $\rightarrow$ paths: Suppose there is an integer-valued flow of value k. Are there k edge-disjoint paths? If so, what are they?
- Construct k edge-disjoint paths from a flow of value $\geq k$.
  - There is an integral flow. Therefore, flow on each edge is 0 or 1.

Mapping to the Max-Flow Problem

- Convert G into a flow network: s is the source, t is the sink, each edge has capacity 1.
- Paths $\rightarrow$ flow: if there are k edge-disjoint paths from s to t, send one unit of flow along each to yield a flow with value k.
- Flow $\rightarrow$ paths: Suppose there is an integer-valued flow of value k. Are there k edge-disjoint paths? If so, what are they?
- Construct k edge-disjoint paths from a flow of value $\geq k$.
  - There is an integral flow. Therefore, flow on each edge is 0 or 1.
  - Claim: if f is a 0-1 valued flow of value ν, then the set of edges with flow $f(e) = 1$ contains a set of ν edge-disjoint paths.

Mapping to the Max-Flow Problem

- Convert G into a flow network: s is the source, t is the sink, each edge has capacity 1.
- Paths $\rightarrow$ flow: if there are k edge-disjoint paths from s to t, send one unit of flow along each to yield a flow with value k.
- Flow $\rightarrow$ paths: Suppose there is an integer-valued flow of value k. Are there k edge-disjoint paths? If so, what are they?
- Construct k edge-disjoint paths from a flow of value $\geq k$.
  - There is an integral flow. Therefore, flow on each edge is 0 or 1.
  - Claim: if f is a 0-1 valued flow of value ν, then the set of edges with flow $f(e) = 1$ contains a set of ν edge-disjoint paths.
  - Prove by induction on the number of edges in f that carry flow.

Mapping to the Max-Flow Problem

- Convert G into a flow network: s is the source, t is the sink, each edge has capacity 1.
- Paths $\rightarrow$ flow: if there are k edge-disjoint paths from s to t, send one unit of flow along each to yield a flow with value k.
- Flow $\rightarrow$ paths: Suppose there is an integer-valued flow of value k. Are there k edge-disjoint paths? If so, what are they?
- Construct k edge-disjoint paths from a flow of value $\geq k$.
  - There is an integral flow. Therefore, flow on each edge is 0 or 1.
  - Claim: if f is a 0-1 valued flow of value ν, then the set of edges with flow $f(e) = 1$ contains a set of ν edge-disjoint paths.
  - Prove by induction on the number of edges in f that carry flow.
- We just proved: there are k edge-disjoint paths from s to t in a directed graph G iff the maximum value of an s-t flow in G is $\geq k$.

Running Time of the Edge-Disjoint Paths Algorithm

- Given a flow of value k, how quickly can we determine the k edge-disjoint paths?

Running Time of the Edge-Disjoint Paths Algorithm

- Given a flow of value k, how quickly can we determine the k edge-disjoint paths? $O(mn)$ time.
- Corollary: The Ford-Fulkerson algorithm can be used to find a maximum set of edge-disjoint s-t paths in a directed graph G in

Running Time of the Edge-Disjoint Paths Algorithm

- Given a flow of value k, how quickly can we determine the k edge-disjoint paths? $O(mn)$ time.
- Corollary: The Ford-Fulkerson algorithm can be used to find a maximum set of edge-disjoint s-t paths in a directed graph G in $O(mn)$ time.

Certificate for Edge-Disjoint Paths Algorithm

- A set $F \subseteq E$ of edge separates s and t if the graph $(V, E - F)$ contains no s-t paths.

Certificate for Edge-Disjoint Paths Algorithm

- A set $F \subseteq E$ of edge separates s and t if the graph $(V, E - F)$ contains no s-t paths.
- Menger's Theorem: In every directed graph with nodes s and t, the maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal disconnects s from t.

Edge-Disjoint Paths in Undirected Graphs

- Can extend the theorem to *undirected* graphs.

Edge-Disjoint Paths in Undirected Graphs

- Can extend the theorem to *undirected* graphs.
- Replace each edge with two directed edges of capacity 1 and apply the algorithm for directed graphs.

Edge-Disjoint Paths in Undirected Graphs

- Can extend the theorem to *undirected* graphs.
- Replace each edge with two directed edges of capacity 1 and apply the algorithm for directed graphs.
- Problem: Both counterparts of an undirected edge (u, v) may be used by different edge-disjoint paths in the directed graph.

Edge-Disjoint Paths in Undirected Graphs

- Can extend the theorem to *undirected* graphs.
- Replace each edge with two directed edges of capacity 1 and apply the algorithm for directed graphs.
- Problem: Both counterparts of an undirected edge (u, v) may be used by different edge-disjoint paths in the directed graph.
- Can obtain an integral flow where only one of the directed counterparts of (u, v) has non-zero flow.

Edge-Disjoint Paths in Undirected Graphs

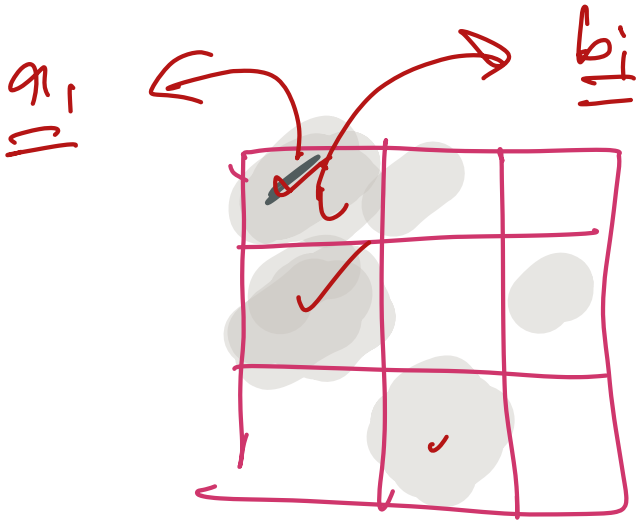
- Can extend the theorem to *undirected* graphs.
- Replace each edge with two directed edges of capacity 1 and apply the algorithm for directed graphs.
- Problem: Both counterparts of an undirected edge (u, v) may be used by different edge-disjoint paths in the directed graph.
- Can obtain an integral flow where only one of the directed counterparts of (u, v) has non-zero flow.
- We can find the maximum number of edge-disjoint paths in $O(mn)$ time.
- We can prove a version of Menger's theorem for undirected graphs: in every undirected graph with nodes s and t, the maximum number of edge-disjoint s–t paths is equal to the minimum number of edges whose removal separates s from t.

Image Segmentation

- A fundamental problem in computer vision is that of segmenting an image into coherent regions.
- A basic segmentation problem is that of partitioning an image into a foreground and a background: label each pixel in the image as belonging to the foreground or the background.

Formulating the Image Segmentation Problem

- Let V be the set of pixels in an image.
- Let E be the set of pairs of neighbouring pixels.
- V and E yield an undirected graph $G(V, E)$.

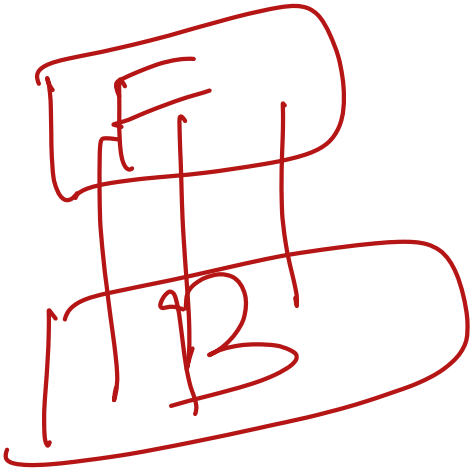


Formulating the Image Segmentation Problem

- Let V be the set of pixels in an image.
- Let E be the set of pairs of neighbouring pixels.
- V and E yield an undirected graph $G(V, E)$.
- Each pixel i has a likelihood $a_i > 0$ that it belongs to the foreground and a likelihood $b_i > 0$ that it belongs to the background.
- These likelihoods are specified in the input to the problem.

Formulating the Image Segmentation Problem

- Let V be the set of pixels in an image.
- Let E be the set of pairs of neighbouring pixels.
- V and E yield an undirected graph $G(V, E)$.
- Each pixel i has a likelihood $a_i > 0$ that it belongs to the foreground and a likelihood $b_i > 0$ that it belongs to the background.
- These likelihoods are specified in the input to the problem.
- We want the foreground/background boundary to be smooth:



Formulating the Image Segmentation Problem

- Let V be the set of pixels in an image.
- Let E be the set of pairs of neighbouring pixels.
- V and E yield an undirected graph $G(V, E)$.
- Each pixel i has a likelihood $a_i > 0$ that it belongs to the foreground and a likelihood $b_i > 0$ that it belongs to the background.
- These likelihoods are specified in the input to the problem.
- We want the foreground/background boundary to be smooth: For each pair (i, j) of pixels, assign separation penalty $p_{ij} \geq 0$ for placing one of them in the foreground and the other in the background.

The Image Segmentation Problem

IMAGE SEGMENTATION

INSTANCE: Pixel graphs $G(V, E)$, likelihood functions $a, b : V \to \mathbb{R}^+$, penalty function $p : E \to \mathbb{R}^+$

SOLUTION: *Optimum labelling*: partition of the pixels into two sets A and B that maximises

$$q(A, B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} \boxed{p_{ij}}.$$

F B

0.7 (F)

0.35 (B)

Developing an Algorithm for Image Segmentation

- There is a similarity between cuts and labellings.
- But there are differences:
  - We are maximising an objective function rather than minimising it.
  - There is no source or sink in the segmentation problem.
  - We have values on the nodes.
  - The graph is undirected.

Maximization to Minimization

- Let $Q = \sum_i (a_i + b_i)$.

Maximization to Minimization

- Let $Q = \sum_i (a_i + b_i)$.
- Notice that $\sum_{i \in A} a_i + \sum_{j \in B} b_j = Q - \sum_{i \in A} b_i + \sum_{j \in B} a_j$.
- Therefore, maximising

$$\begin{aligned} q(A,B) &= \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cup \{i,j\}| = 1}} p_{ij} \\ &= Q - \sum_{i \in A} b_i - \sum_{j \in B} a_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij} \end{aligned}$$

is identical to minimising

$$q'(A,B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}$$

mini

Solving the Other Issues

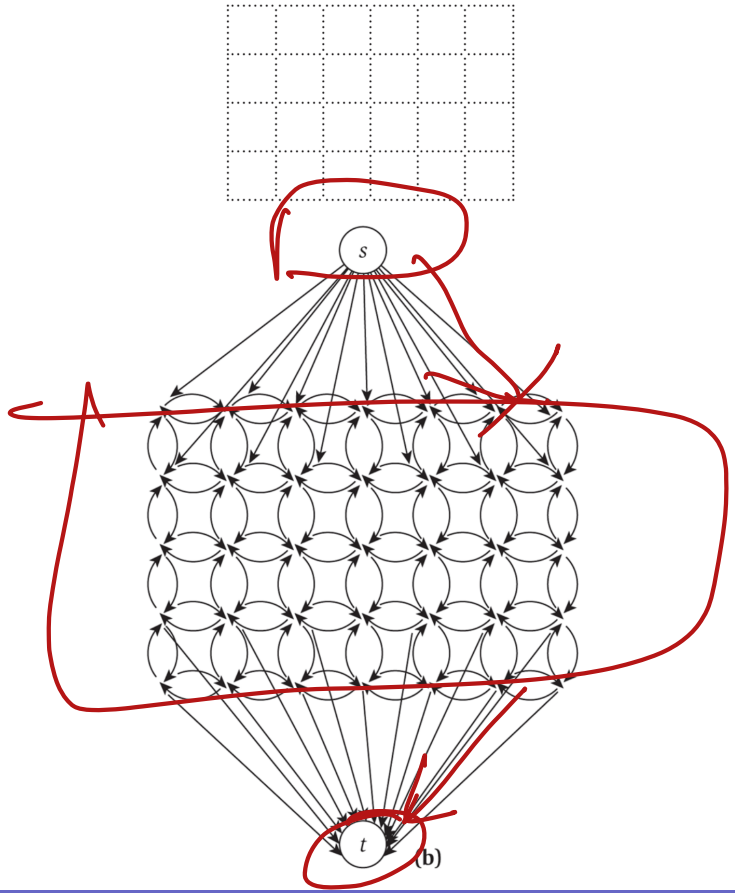
- Solve the issues like we did earlier.

Solving the Other Issues

- Solve the issues like we did earlier.
- Add a new "super-source" s to represent the foreground.
- Add a new "super-sink" t to represent the background.

Solving the Other Issues

- Solve the issues like we did earlier.
- Add a new "super-source" s to represent the foreground.
- Add a new "super-sink" t to represent the background.
- Connect s and t to every pixel and assign capacity a_i to edge (s, i) and capacity b_i to edge (i, t).
- Direct edges away from s and into t.
- Replace each edge (i, j) in E with two directed edges of capacity p_{ij}.



Cuts in the Flow Network

- Let G' be this flow network and (A, B) an s-t cut.
- What does the capacity of the cut represent?

Cuts in the Flow Network

- Let G' be this flow network and (A, B) an s-t cut.
- What does the capacity of the cut represent?
- Edges crossing the cut are of three types:

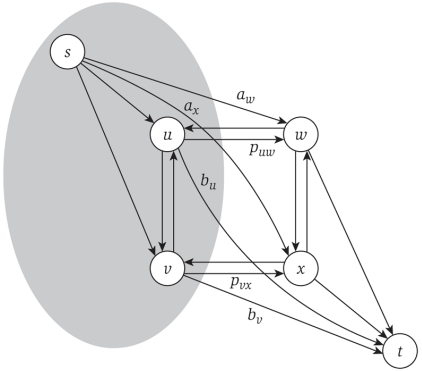


Figure 7.19 An s-t cut on a graph constructed from four pixels. Note how the three types of terms in the expression for $q'(A, B)$ are captured by the cut.

Cuts in the Flow Network

- Let G' be this flow network and (A, B) an s-t cut.
- What does the capacity of the cut represent?
- Edges crossing the cut are of three types:
  - $(s, w), w \in B$ contributes a_w.
  - $(u, t), u \in A$ contributes b_u.
  - $(u, w), u \in A, w \in B$ contributes p_{uw}.

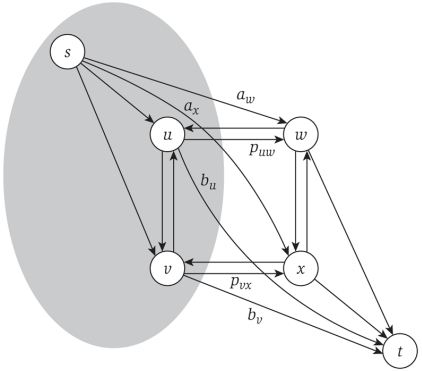


Figure 7.19 An s-t cut on a graph constructed from four pixels. Note how the three types of terms in the expression for $q'(A, B)$ are captured by the cut.

Cuts in the Flow Network

- Let G' be this flow network and (A, B) an s-t cut.
- What does the capacity of the cut represent?
- Edges crossing the cut are of three types:
  - $(s, w), w \in B$ contributes a_w.
  - $(u, t), u \in A$ contributes b_u.
  - $(u, w), u \in A, w \in B$ contributes p_{uw}.

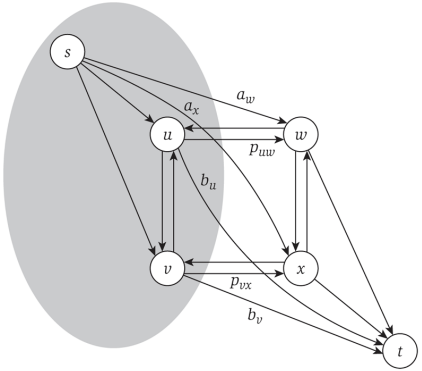


Figure 7.19 An s-t cut on a graph constructed from four pixels. Note how the three types of terms in the expression for $q'(A, B)$ are captured by the cut.

$$c(A, B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij} = q'(A, B).$$

Solving the Image Segmentation Problem

- The capacity of a s-t cut $c(A, B)$ exactly measures the quantity $q'(A, B)$.
- To maximise $q(A, B)$, we simply compute the s-t cut (A, B) of minimum capacity.
- Deleting s and t from the cut yields the desired segmentation of the image.

AKS Algorithm → Primality Test

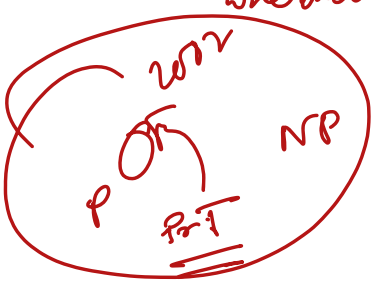
Polynomial time Algorithm / Efficient

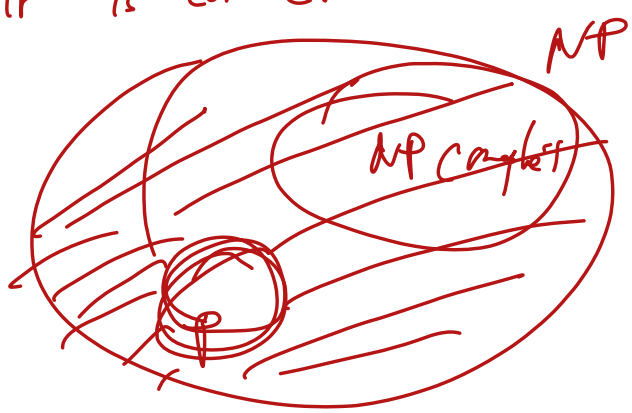
Optimization / Search Problem

$\boxed{P = NP}$?

P → You have some polynomial algorithm to give you the correct solution.

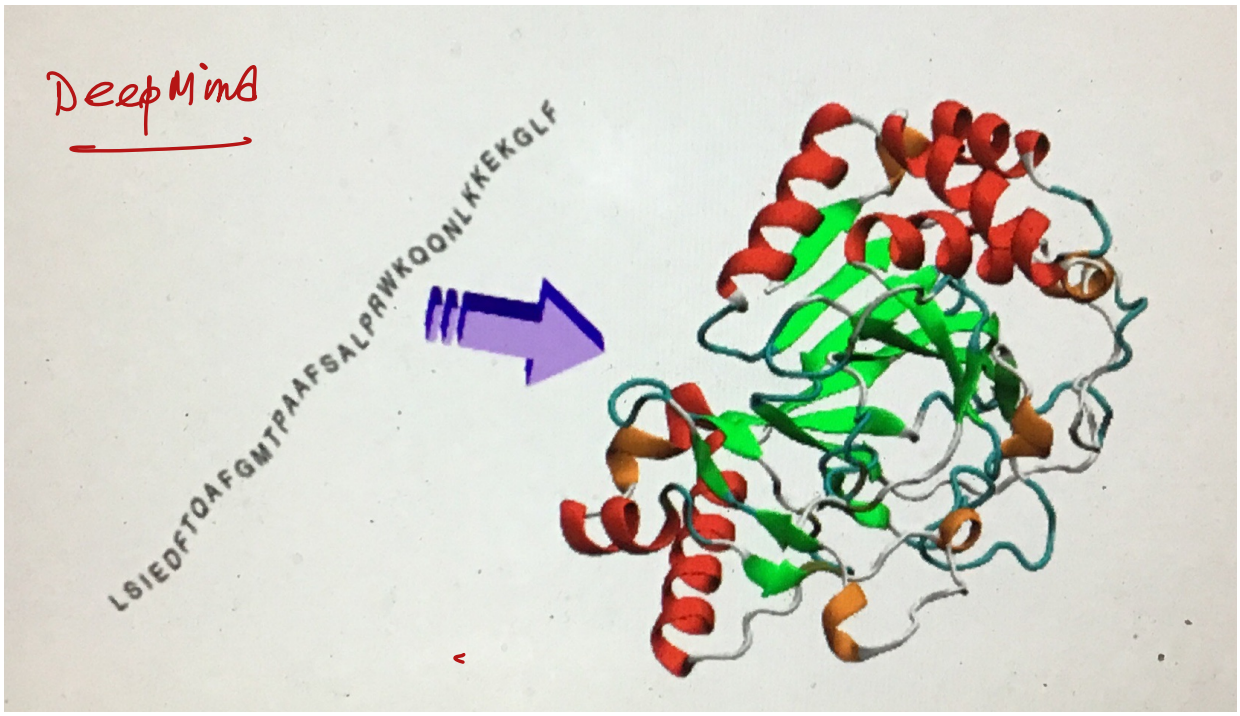
NP → you can verify given a solution, whether it is correct





More Hard Computational Problems

Aerospace engineering: optimal mesh partitioning for finite elements.
Biology: protein folding.
Chemical engineering: heat exchanger network synthesis.
Civil engineering: equilibrium of urban traffic flow.
Economics: computation of arbitrage in financial markets with friction.
Electrical engineering: VLSI layout.
Environmental engineering: optimal placement of contaminant sensors.
Financial engineering: find minimum risk portfolio of given return.
Game theory: find Nash equilibrium that maximizes social welfare.
Genomics: phylogeny reconstruction.
Mechanical engineering: structure of turbulence in sheared flows.
Medicine: reconstructing 3-D shape from biplane angiocardiogram.
Operations research: optimal resource allocation.
Physics: partition function of 3-D Ising model in statistical mechanics.
Politics: Shapley-Shubik voting power.
Pop culture: Minesweeper consistency.
Statistics: optimal experimental design.



Motif finding problem parameters

L=35

n=5

tacatAGAAGAAAGGggcgtacacacgttacgccg

tttgagcagatttagtcctggaaaCAATAAAACGa

tgggatgacttAAAATAATGGtgcggatcattcga

ggatgCAAAAAAAGGtccacgcaaaggcaaggaga

ggtaaggctggttacgtagTAATAAAGGctatag

AGAAGAAAGG
CAATAAAACG
AAAATAATGG
CAAAAAAAGG
ATAATAAAGG

$$f = \frac{1}{5}\begin{bmatrix} 3&4&5&4&2&5&5&4&0&0 \\ 2&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&1&0&0&0&4&5 \\ 0&1&0&1&2&0&0&1&0&0 \end{bmatrix} \begin{matrix} A \\ C \\ G \\ T \end{matrix}$$

Position weight matrix (PWM

Outline

1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

Polynomial vs Exponential

running time:	n	n^2	n^3	2^n
less than 10^9:	10^9	$10^{4.5}$	10^3	29

30

Improving Brute Force Search

Usually, an efficient (polynomial) algorithm searches for a solution among an exponential number of candidates:

- there are $n!$ permutations of n objects

Improving Brute Force Search

Usually, an efficient (polynomial) algorithm searches for a solution among an exponential number of candidates:

- there are $n!$ permutations of n objects
- there are 2^n ways to partition n objects into two sets

A*

Heuristic

GA, SA, PSO . — —

Improving Brute Force Search

Usually, an efficient (polynomial) algorithm searches for a solution among an exponential number of candidates:

- there are $n!$ permutations of n objects
- there are 2^n ways to partition n objects into two sets
- there are n^{n-2} spanning trees in a complete graph on n vertices

This module

- For thousands of practically important problems we don't have an efficient algorithm yet

This module

- For thousands of practically important problems we don't have an efficient algorithm yet
- An efficient algorithm for one such problem automatically gives efficient algorithms for all these problems!

This module

- For thousands of practically important problems we don't have an efficient algorithm yet
- An efficient algorithm for one such problem automatically gives efficient algorithms for all these problems!
- $1M prize for constructing such an algorithm or proving that this is impossible!

Outline

1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

$\vee \to$ or
$\wedge$ and

Boolean Formulas

Formula in conjunctive normal form

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z}) \to \text{True}$$

- x, y, z are Boolean variables (values: `true`/`false` or 1/0)
- literals are variables (x, y, z) and their negations $(\overline{x}, \overline{y}, \overline{z})$
- clauses are disjunctions (logical or) of literals

Satisfiability (SAT)

Input: Formula F in conjunctive normal form (CNF).

Output: An assignment of Boolean values to the variables of F satisfying all clauses, if exists.

$(\quad \vee \quad \vee \quad) \wedge (\qquad\qquad) \wedge (\quad)$

Examples

CNF $x \vee \bar{y}$
$=1 \vee \bar{0} = 1$

- The formula $(x \vee \overline{y})(\overline{x} \vee \overline{y})(x \vee y)$ is satisfiable: set $x = 1$, $y = 0$.
- The formula $(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})$ is satisfiable: set $x = 1, y = 1, z = 1$ or $x = 1, y = 0, z = 0$.
- The formula $(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$ is unsatisfiable.

Satisfiability

- Classical hard problem
- Many applications: e.g., hardware/software verification, planning, scheduling
- Many hard problems are stated in terms of SAT naturally
- SAT solvers (will see later), SAT competition

- SAT is a typical search problem
- Search problem: given an instance I, find a solution S or report that none exists
- Main property: one must be able to check quickly whether S is indeed a solution for I
- By saying quickly, we mean in time polynomial in the length of I. In particular, the length of S should be polynomial in the length of I

Definition

A search problem is defined by an algorithm $\mathcal{C}$ that takes an instance I and a candidate solution S, and runs in time polynomial in the length of I. We say that S is a solution to I iff $\mathcal{C}(S, I) = \texttt{true}$.

NP $\rightarrow$ Verify in Polynomial time

Example

CNF

For SAT, I is a Boolean formula, S is an assignment of Boolean constants to its variables. The corresponding algorithm $\mathcal{C}$ checks whether S satisfies all clauses of I.

Next part

A few practical search problems for which polynomial algorithms remain unknown

Outline

1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

Outline

1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

Traveling salesman problem (TSP)

Input: Pairwise distances between n cities and a budget b.

Output: A cycle that visits each vertex exactly once and has total length at most b.

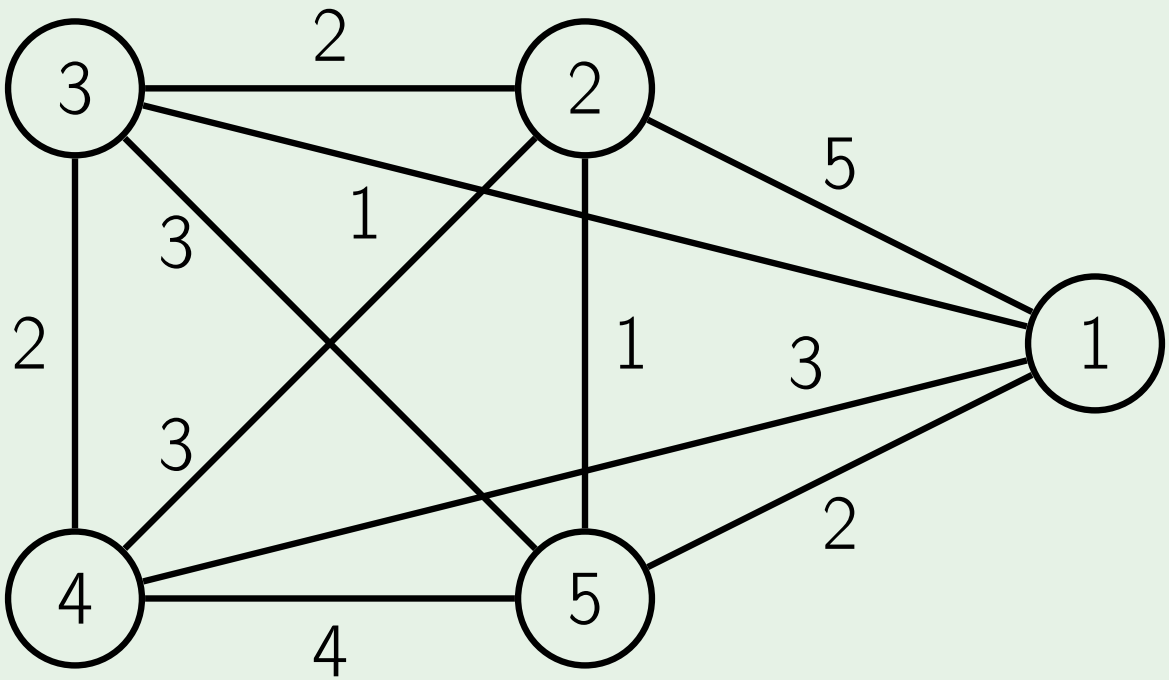
Delivery Company

https://simple.wikipedia.org/wiki/Travelling_salesman_problem

Drilling Holes in a Circuit Board

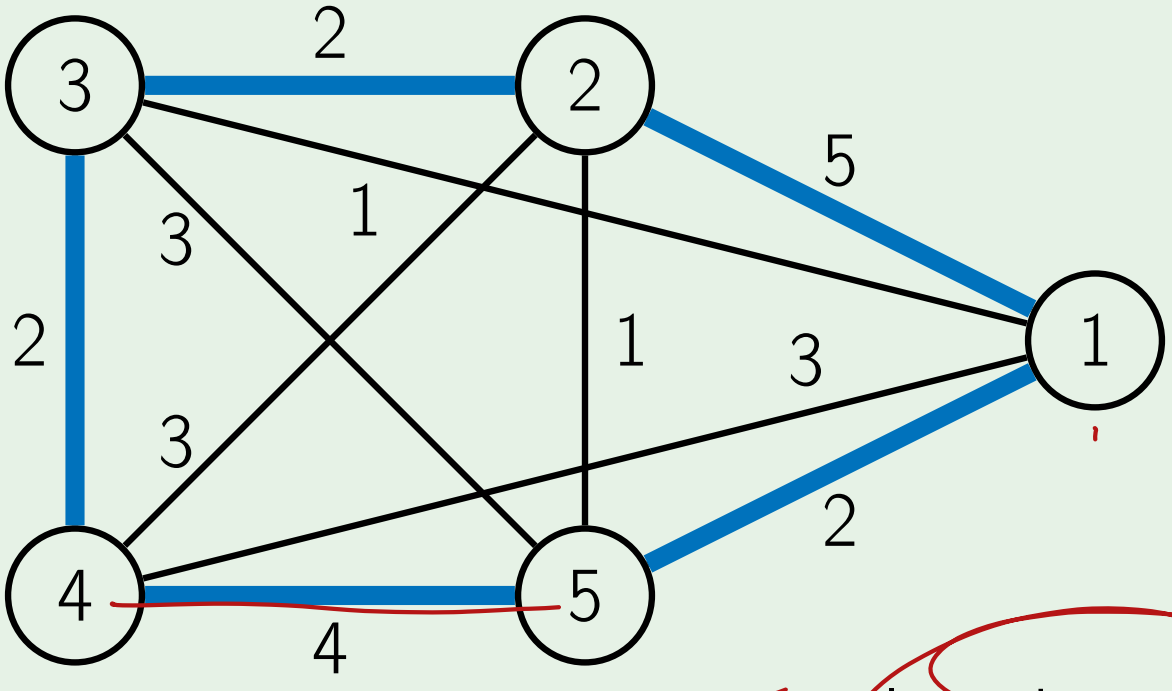
https://developers.google.com/optimization/routing/tsp

Example



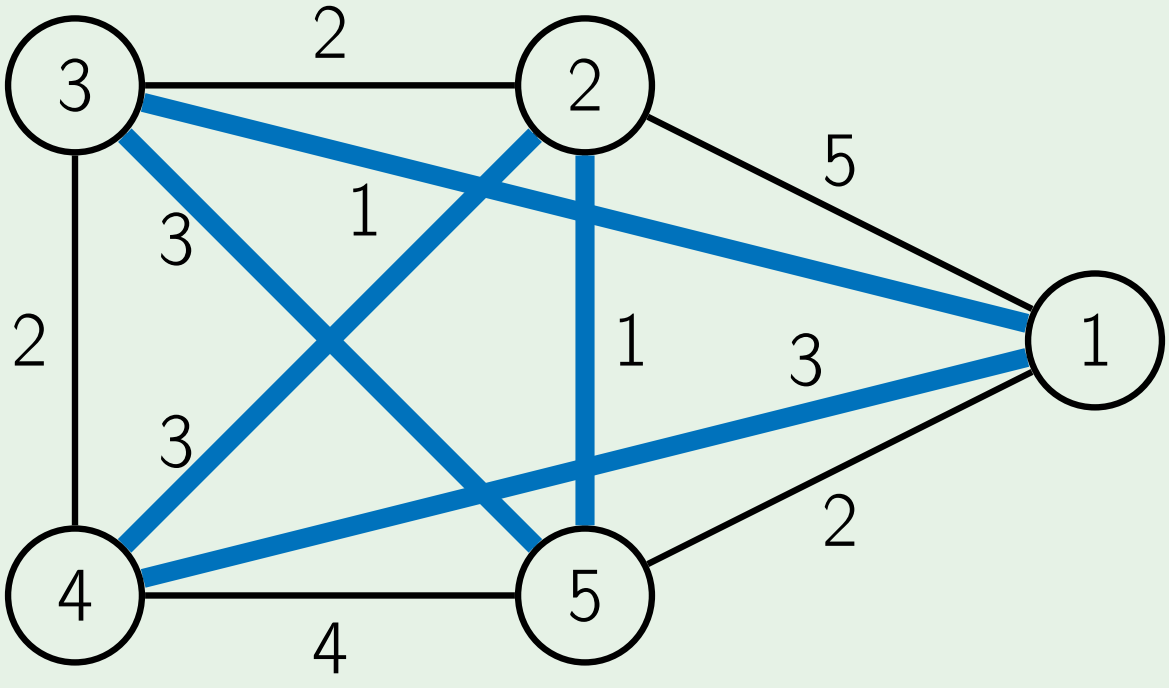
Example

Branch bound



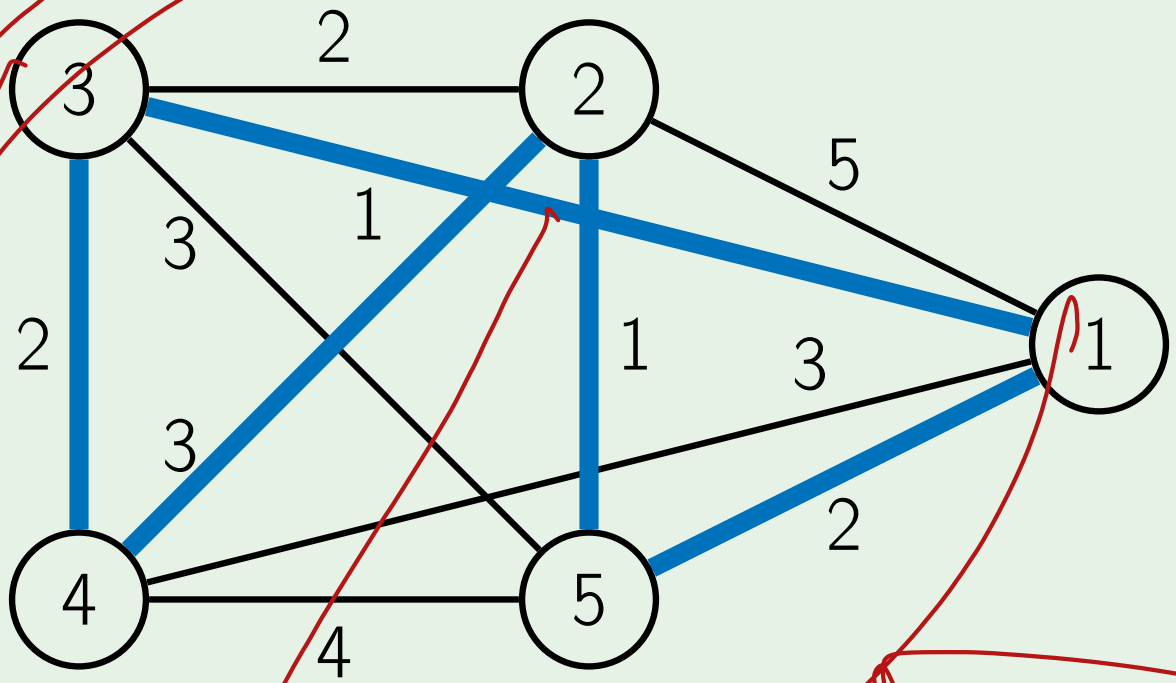
length: 15

Example



length: 13

Example



length: 9

Search Problem

- TSP is a search problem: given a sequence of vertices, it is easy to check whether it is a cycle visiting all the vertices of total length at most b
- TSP is usually stated as an optimization problem; we stated its decision version to guarantee that a candidate solution can be efficiently checked for correctness

Algorithms

- Check all permutations: about $O(n!)$, extremely slow
- Dynamic programming: $O(n^2 2^n)$
- No significantly better upper bound is known
- There are heuristic algorithms and approximation algorithms

Comparing to MST

MST

Decision version:

given n cities, connect them by $(n-1)$ roads of minimal total length

Comparing to MST

MST

Decision version:
given n cities, connect them by $(n-1)$ roads of minimal total length

Can be solved efficiently

Comparing to MST

MST

Decision version:

given n cities, connect them by $(n-1)$ roads of minimal total length

TSP

Decision version:

given n cities, connect them in a path by $(n-1)$ roads of minimal total length

Can be solved efficiently

Comparing to MST

MST	TSP
Decision version: given n cities, connect them by $(n-1)$ roads of minimal total length	Decision version: given n cities, connect them in a path by $(n-1)$ roads of minimal total length
Can be solved efficiently	No polynomial algorithm known!

Outline

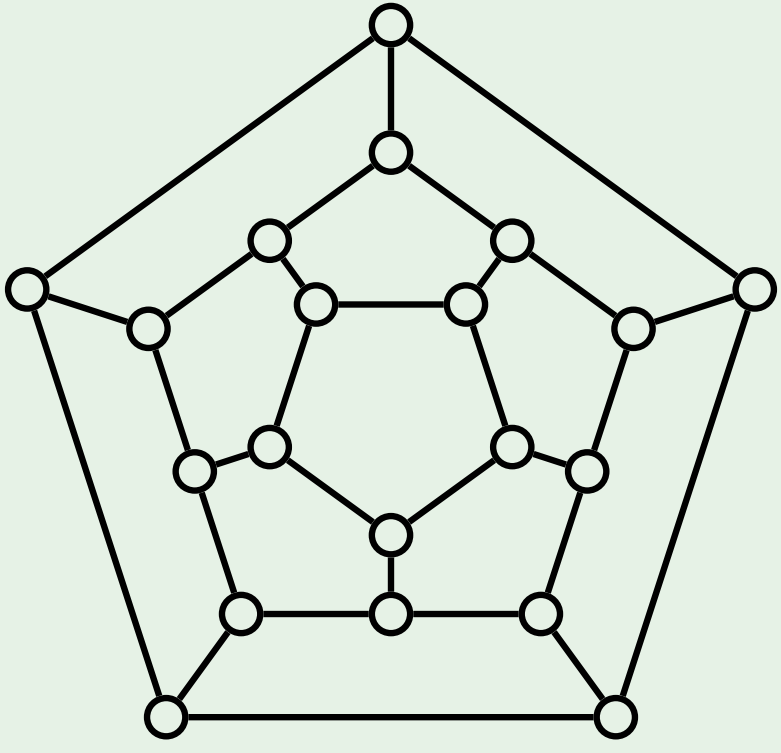
1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

Hamiltonian cycle

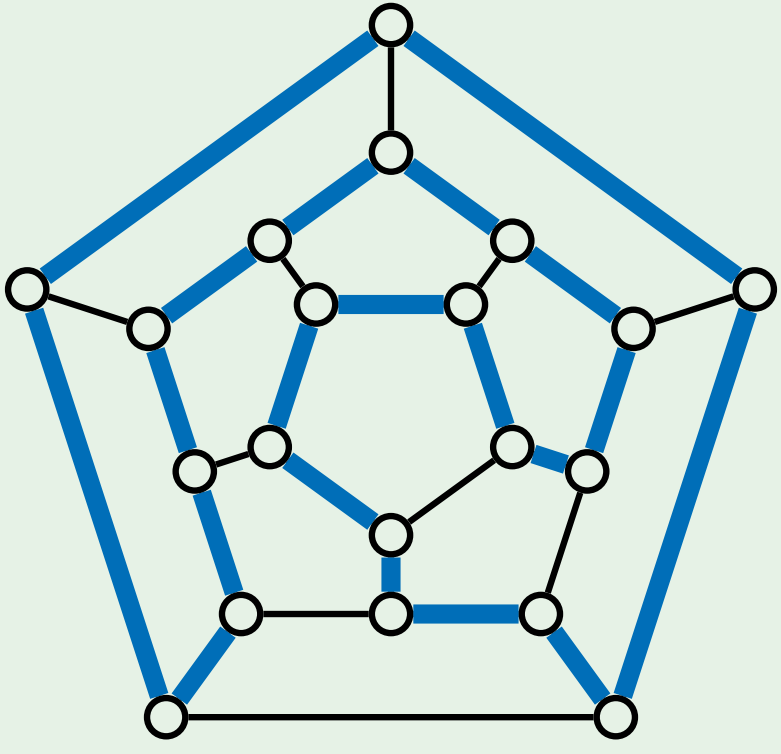
Input: A graph.

Output: A cycle that visits each vertex of the graph exactly once.

Example



Example



Eulerian cycle

Input: A graph.

Output: A cycle that visits each edge of the graph exactly once.

Eulerian cycle

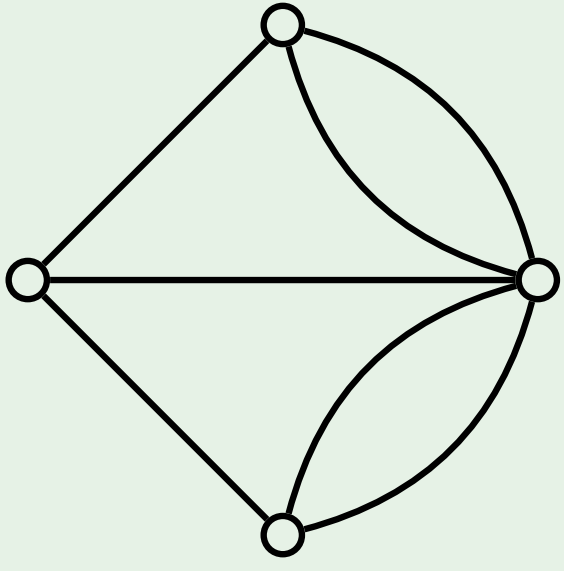
Input: A graph.

Output: A cycle that visits each edge of the graph exactly once.

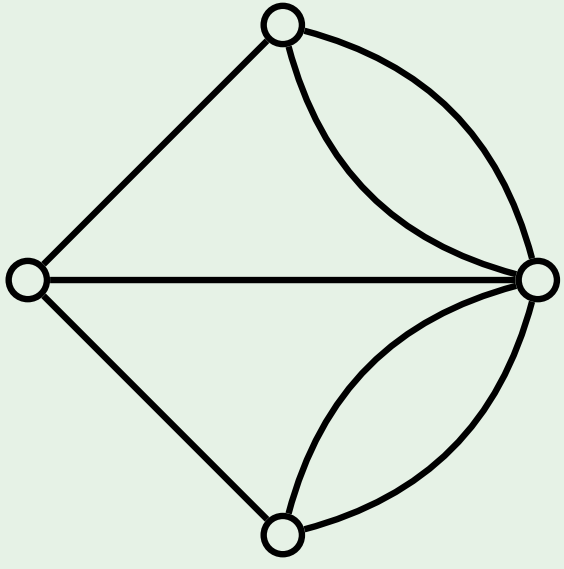
Theorem

A graph has an Eulerian cycle if and only if it is connected and the degree of each vertex is even.

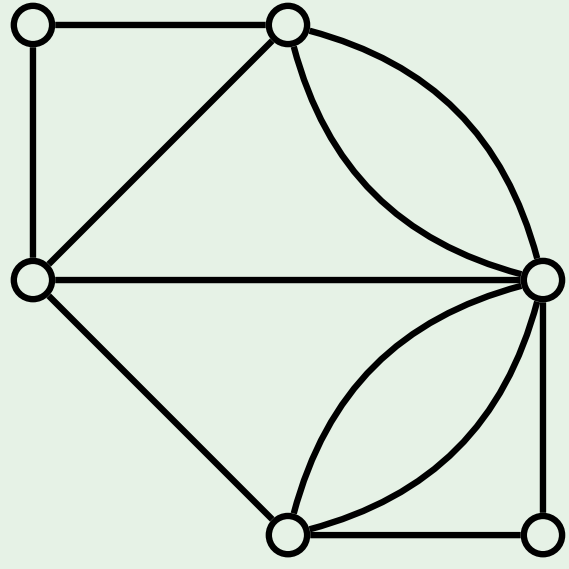
Non-Eulerian graph



Non-Eulerian graph



Eulerian graph



Eulerian cycle

Find a cycle visiting each edge exactly once

Eulerian cycle

Find a cycle visiting each edge exactly once

Can be solved efficiently

Eulerian cycle

Find a cycle visiting each edge exactly once

Hamiltonian cycle

Find a cycle visiting each vertex exactly once

Can be solved efficiently

Eulerian cycle

Find a cycle visiting each edge exactly once

Hamiltonian cycle

Find a cycle visiting each vertex exactly once

Can be solved efficiently

No polynomial algorithm known!

Outline

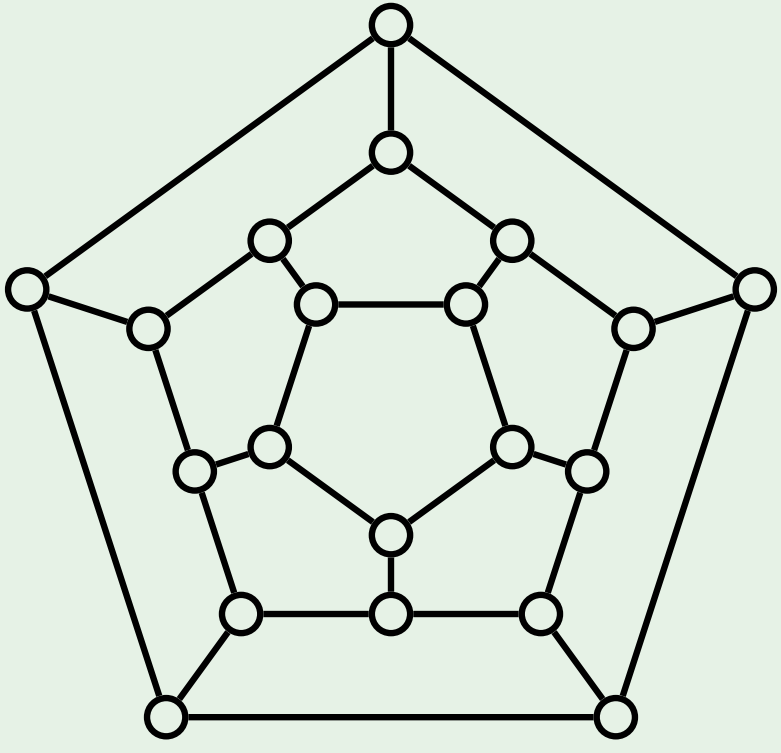
1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

Longest path

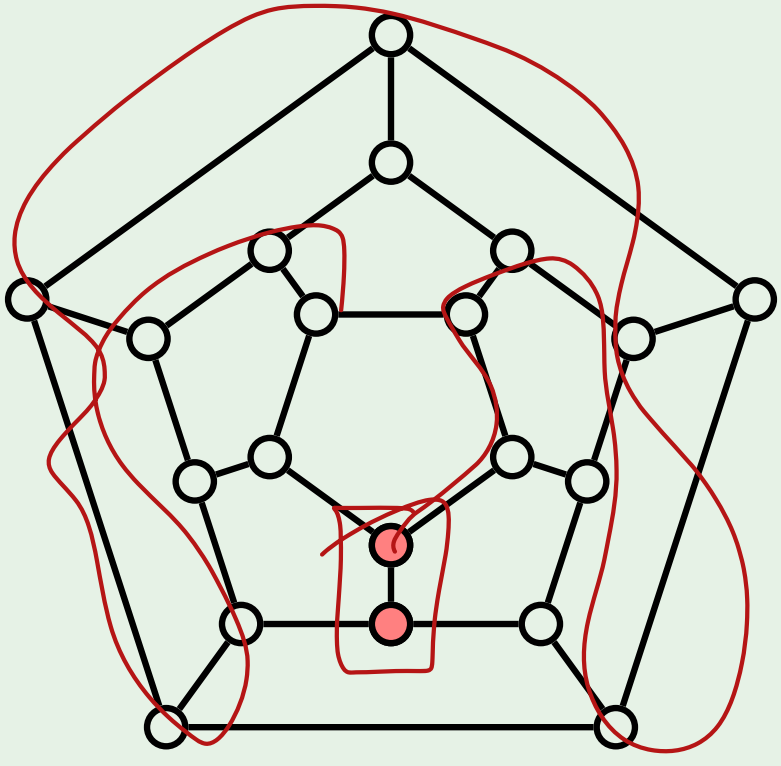
Input: A weighted graph, two vertices s, t, and a budget b.

Output: A simple path (containing no repeated vertices) of total length at least b.

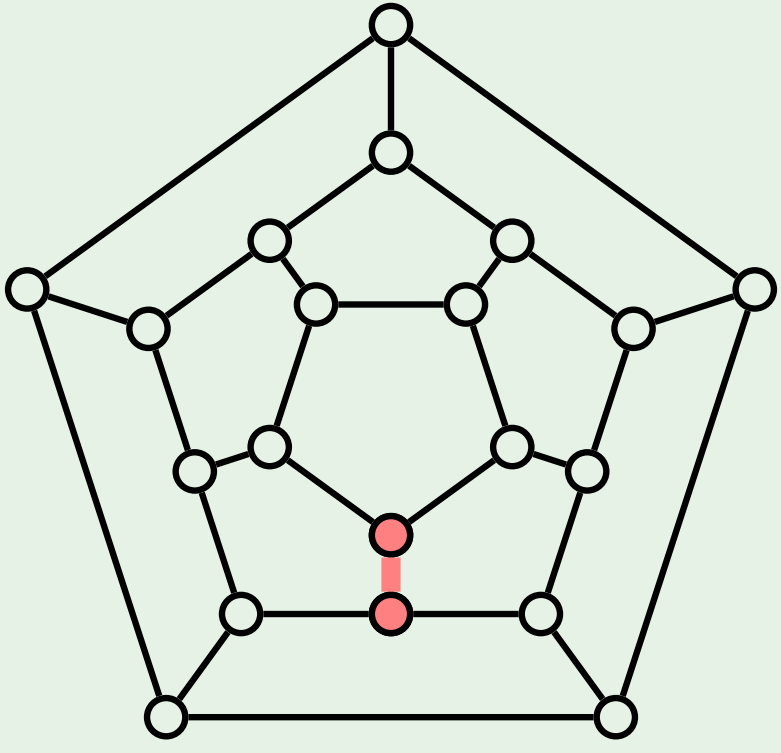
Example



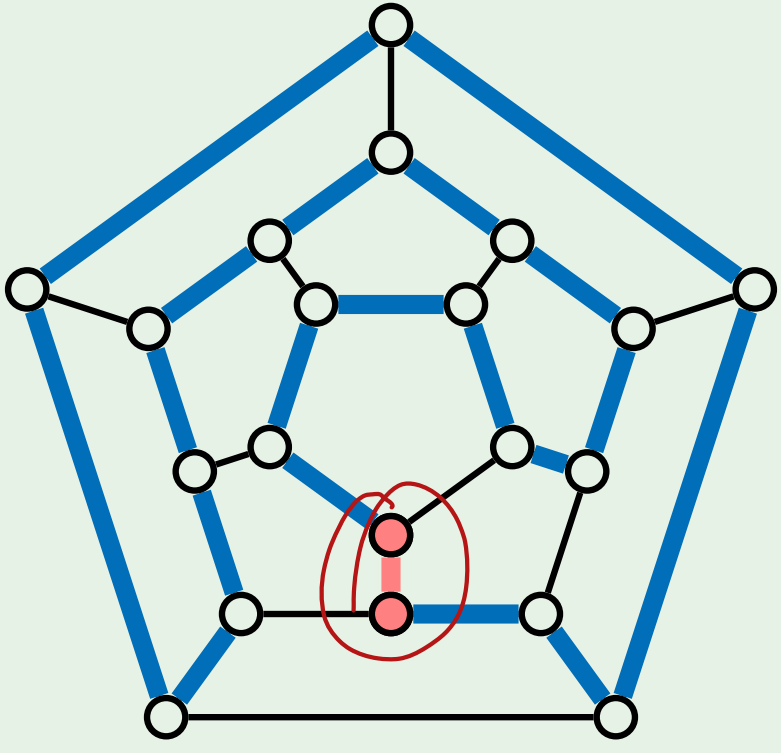
Example



Example



Example



Shortest path

Find a simple path from s to t of total length at most b

Shortest path

Find a simple path from s to t of total length at most b

Can be solved efficiently

Shortest path

Find a simple path from s to t of total length at most b

Longest path

Find a simple path from s to t of total length at least b

Can be solved efficiently

Shortest path

Find a simple path from s to t of total length at most b

Longest path

Find a simple path from s to t of total length at least b

Can be solved efficiently

No polynomial algorithm known!

Outline

1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

Integer linear programming

Input: A set of linear inequalities $\mathbf{Ax} \leq \mathbf{b}$.

Output: Integer solution.

Example

$$x_1 - x_2 \leq c$$

$$x_1 \geq 0.5$$

$$-x_1 + 8x_2 \geq 0$$

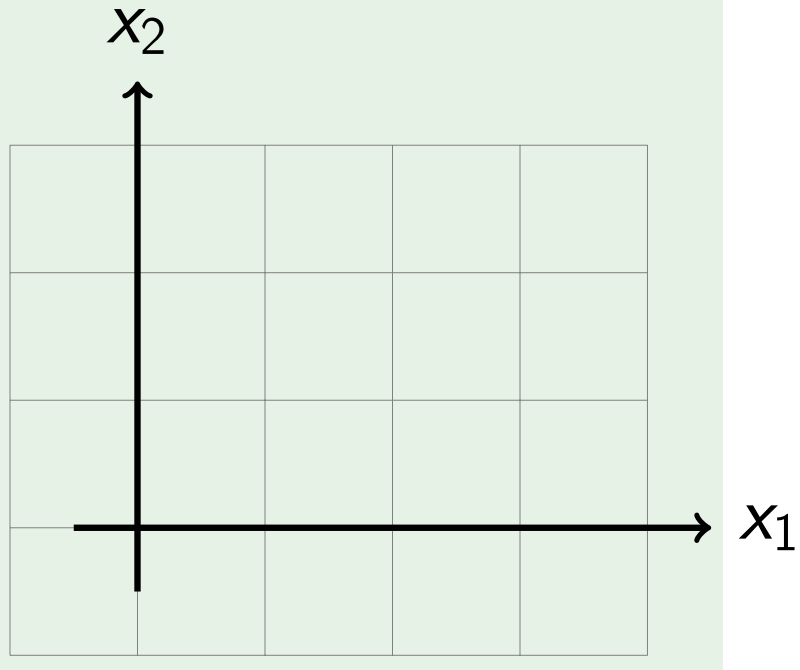
$$-x_1 - 8x_2 \geq -8$$

Example

$$x_1 \geq 0.5$$

$$-x_1 + 8x_2 \geq 0$$

$$-x_1 - 8x_2 \geq -8$$

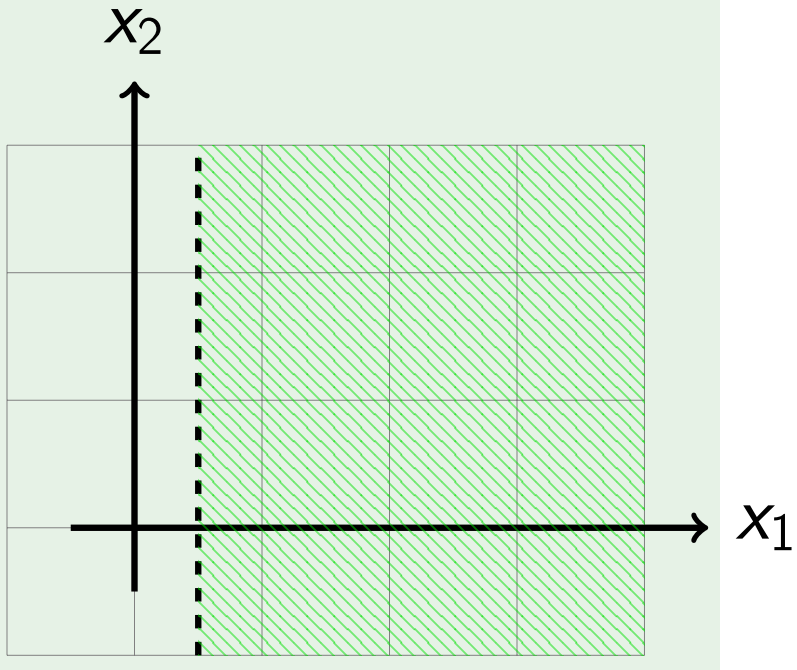


Example

$$x_1 \geq 0.5$$

$$-x_1 + 8x_2 \geq 0$$

$$-x_1 - 8x_2 \geq -8$$

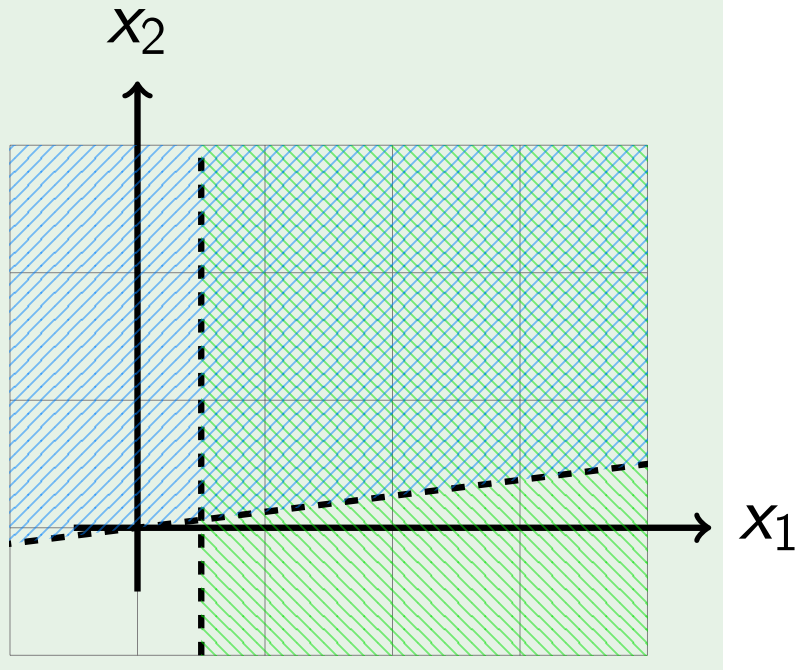


Example

$$x_1 \geq 0.5$$

$$-x_1 + 8x_2 \geq 0$$

$$-x_1 - 8x_2 \geq -8$$

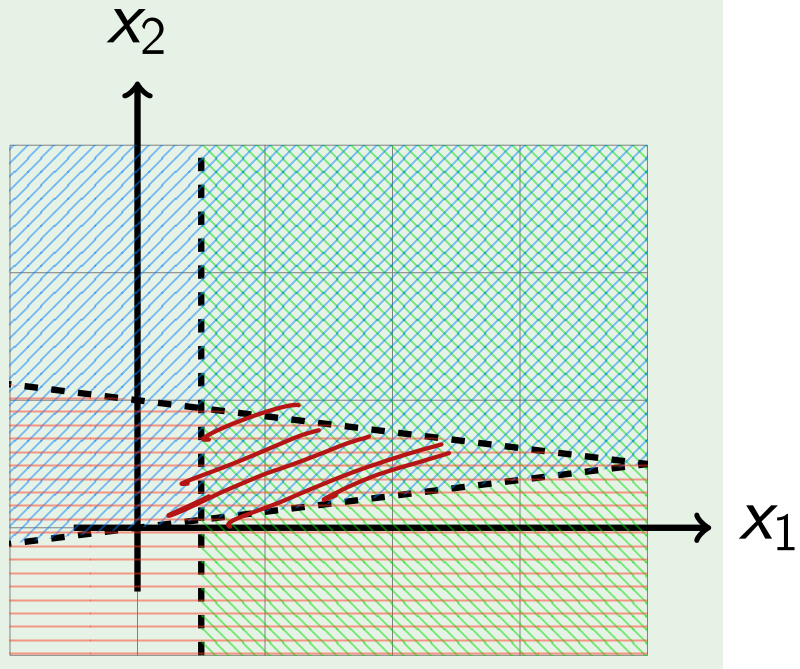


Example

$$x_1 \geq 0.5$$

$$-x_1 + 8x_2 \geq 0$$

$$-x_1 - 8x_2 \geq -8$$



LP (decision version)

Find a real solution of a system of linear inequalities

LP (decision version)

Find a real solution of a system of linear inequalities

Can be solved efficiently

LP (decision version)	ILP
Find a real solution of a system of linear inequalities	Find an integer solution of a system of linear inequalities
Can be solved efficiently	

LP (decision version)	ILP
Find a real solution of a system of linear inequalities	Find an integer solution of a system of linear inequalities
Can be solved efficiently	No polynomial algorithm known!

Outline

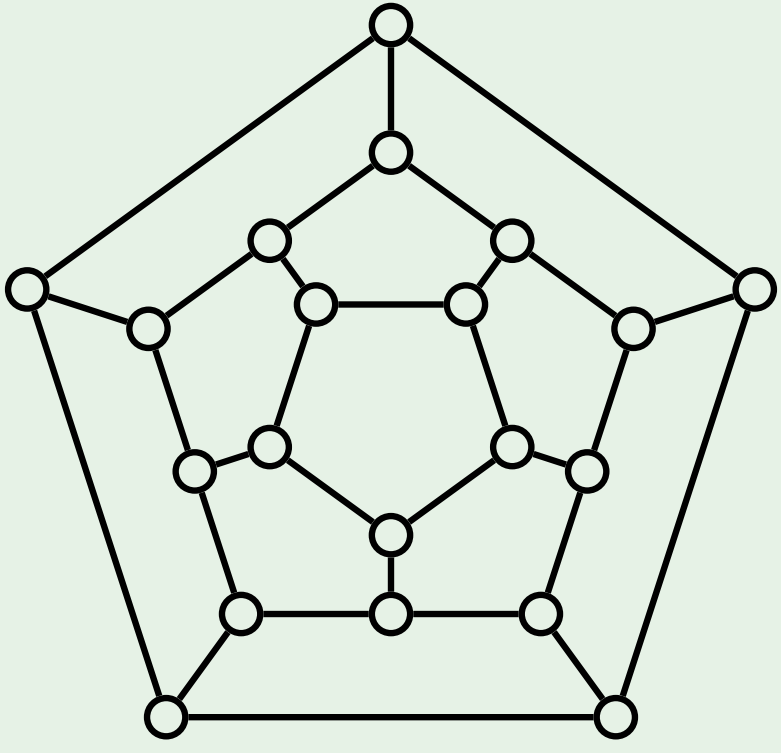
1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

Independent set

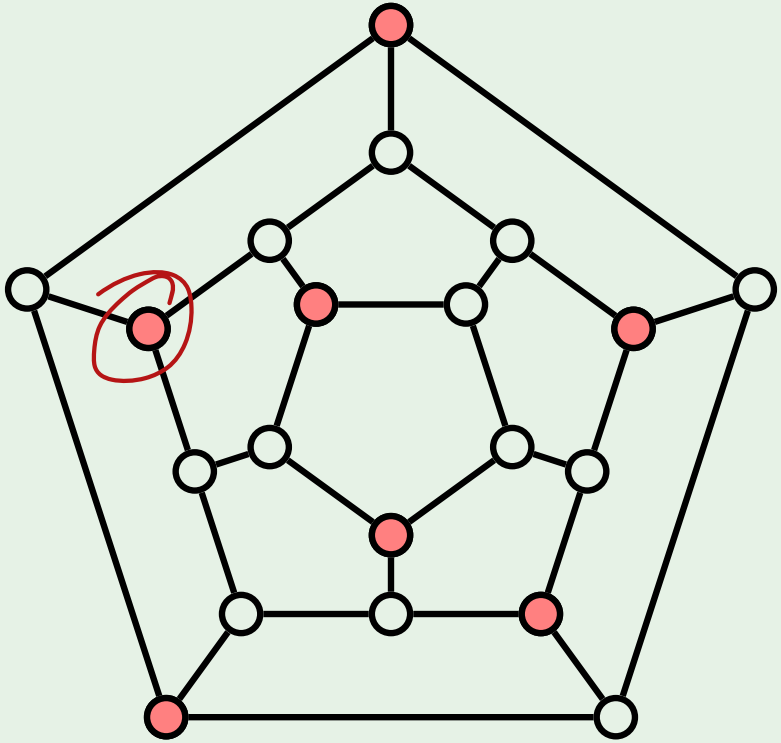
Input: A graph and a budget b.

Output: A subset of vertices of size at least b such that no two of them are adjacent.

Example

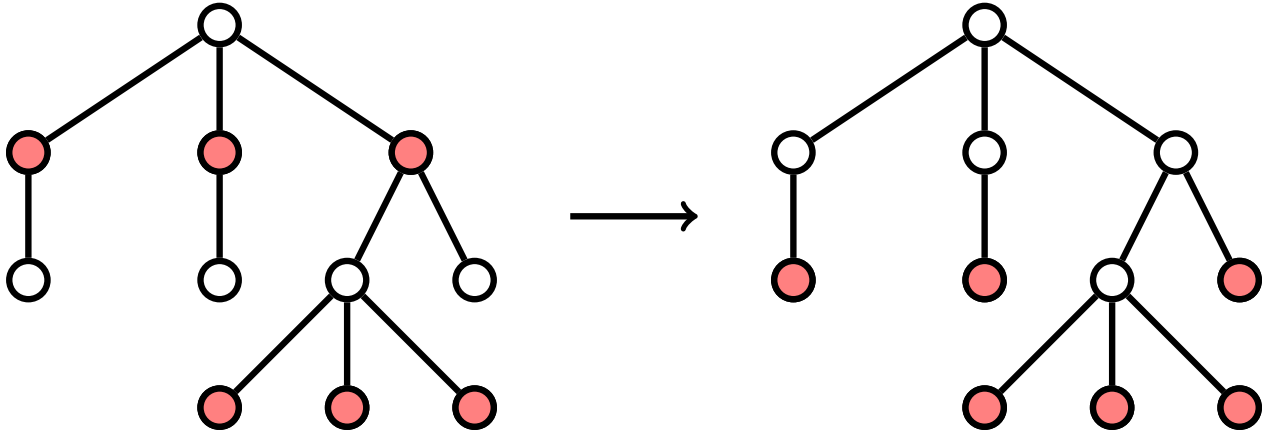


Example



Independent Sets in a Tree

A maximal independent set in a tree can be found by a simple greedy algorithm: it is safe to take into a solution all the leaves.



Independent set in a tree

Find an independent set of size at least b in a given tree

Independent set in a tree

Find an independent set of size at least b in a given tree

Can be solved efficiently

Independent set in a tree

Find an independent set of size at least b in a given tree

Independent set in a graph

Find an independent set of size at least b in a given graph

Can be solved efficiently

Independent set in a tree

Find an independent set of size at least b in a given tree

Can be solved efficiently

Independent set in a graph

Find an independent set of size at least b in a given graph

No polynomial algorithm known!

Next part

It turns out that all these hard problems are in a sense a single hard problem:
a polynomial time algorithm for any of these problems can be used to solve all of them in polynomial time!

Outline

1. Brute Force Search
2. Search Problems
3. Easy and Hard Problems
   - Traveling Salesman Problem
   - Hamiltonian Cycle Problem
   - Longest Path Problem
   - Integer Linear Programming Problem
   - Independent Set Problem
4. P and NP

Class **NP**

Definition

A search problem is defined by an algorithm $\mathcal{C}$ that takes an instance I and a candidate solution S, and runs in time polynomial in the length of I. We say that S is a solution to I iff $\mathcal{C}(S, I) = \texttt{true}$.

Class **NP**

Definition

A search problem is defined by an algorithm $\mathcal{C}$ that takes an instance I and a candidate solution S, and runs in time polynomial in the length of I. We say that S is a solution to I iff $\mathcal{C}(S, I) = \texttt{true}$.

Definition

NP is the class of all search problems.

- **NP** stands for "non-deterministic polynomial time": one can guess a solution, and then verify its correctness in polynomial time
- In other words, the class **NP** contains all problems whose solutions can be efficiently verified

Class **P**

Definition

P is the class of all search problems that can be solved in polynomial time.

Class P

Problems whose solution can be found efficiently

Class P

Problems whose solution can be found efficiently

- MST
- Shortest path
- LP
- IS on trees

Class **P**

Problems whose solution can be found efficiently

Class **NP**

Problems whose solution can be verified efficiently

- MST
- Shortest path
- LP
- IS on trees

Class **P**

Problems whose solution can be found efficiently

- MST
- Shortest path
- LP
- IS on trees

Class **NP**

Problems whose solution can be verified efficiently

- TSP
- Longest path
- ILP
- IS on graphs

The main open problem in Computer Science

Is **P** equal to **NP**?

The main open problem in Computer Science

Is **P** equal to **NP**?

Millenium Prize Problem

Clay Mathematics Institute: $1M prize for solving the problem

- If $\mathbf{P}=\mathbf{NP}$, then all search problems can be solved in polynomial time.

- If $\mathbf{P}=\mathbf{NP}$, then all search problems can be solved in polynomial time.
- If $\mathbf{P}\neq\mathbf{NP}$, then there exist search problems that cannot be solved in polynomial time.

Next part

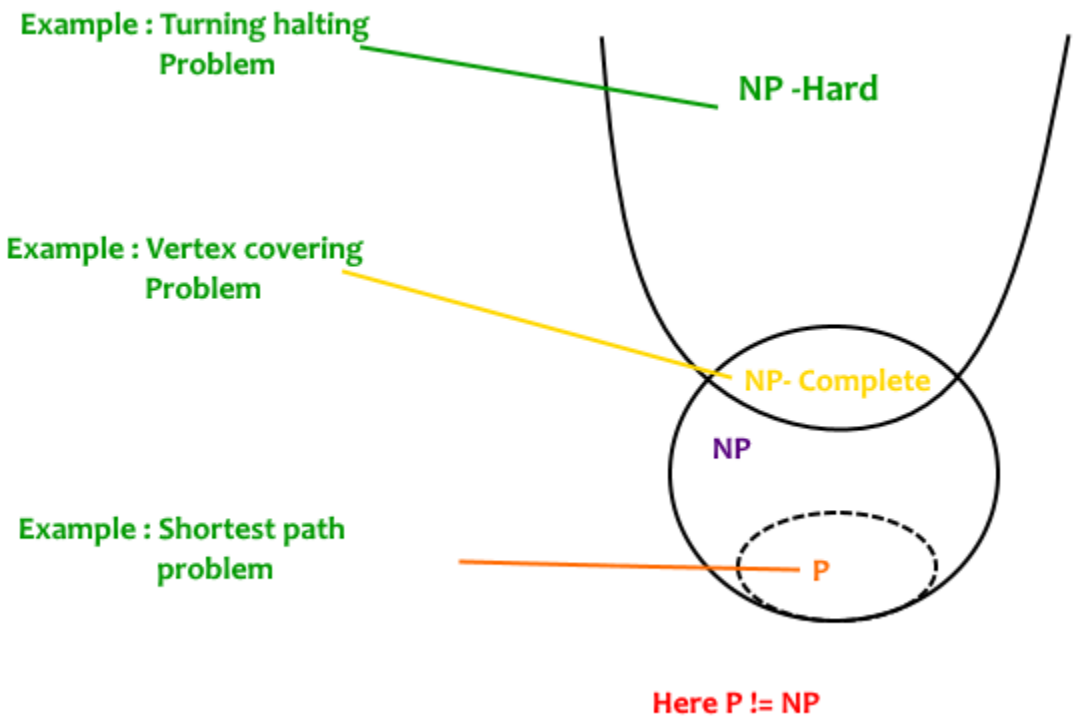
We'll show that the satisfiability problem, the traveling salesman problem, the independent set problem, the integer linear programming are the hardest problems in **NP**.

NP

The lectures are largely inspired by this course

- https://www.coursera.org/learn/advanced-algorithms-and-complexity/lecture/oaA17/all-of-np-to-circuit-sat
- You will get all conversions taught in the last class
- Independent set, vertex cover, SAT and 3-SAT
- Quite well explained.

NP-complete


Example : Turning halting Problem
NP -Hard
Example : Vertex covering Problem
NP- Complete
NP
Example : Shortest path problem
P
Here P != NP

NP-hard class

An NP-hard problem is at least as hard as the hardest problem in NP and it is a class of problems such that every problem in NP reduces to NP-hard.

Features:

1. All NP-hard problems are not in NP.
2. It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.
3. A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

Some of the examples of problems in Np-hard are:

1. **Halting problem.**
2. **Qualified Boolean formulas.**
3. **No Hamiltonian cycle.**

NP-complete class

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

Features:

1. NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
2. If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

Some example problems include:

1. **Decision version of 0/1 Knapsack.**
2. **Hamiltonian Cycle.**
3. **Satisfiability.**
4. **Vertex cover.**

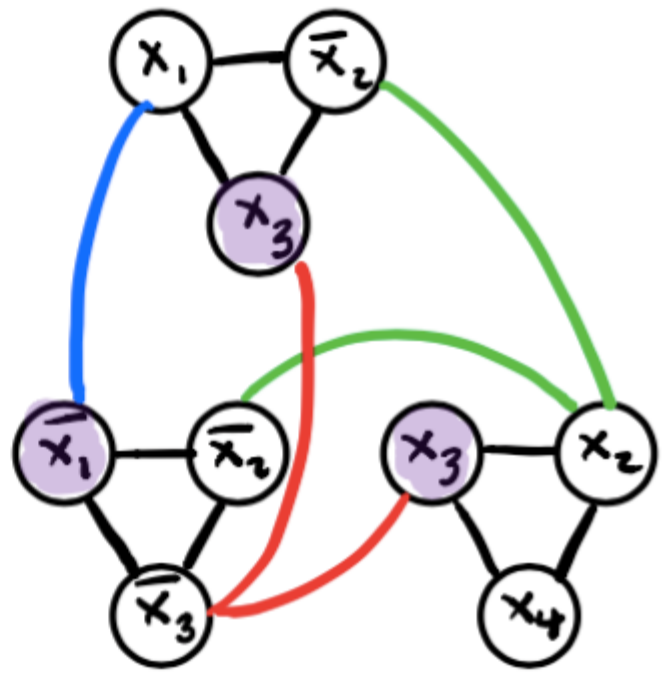
$f \to f'$

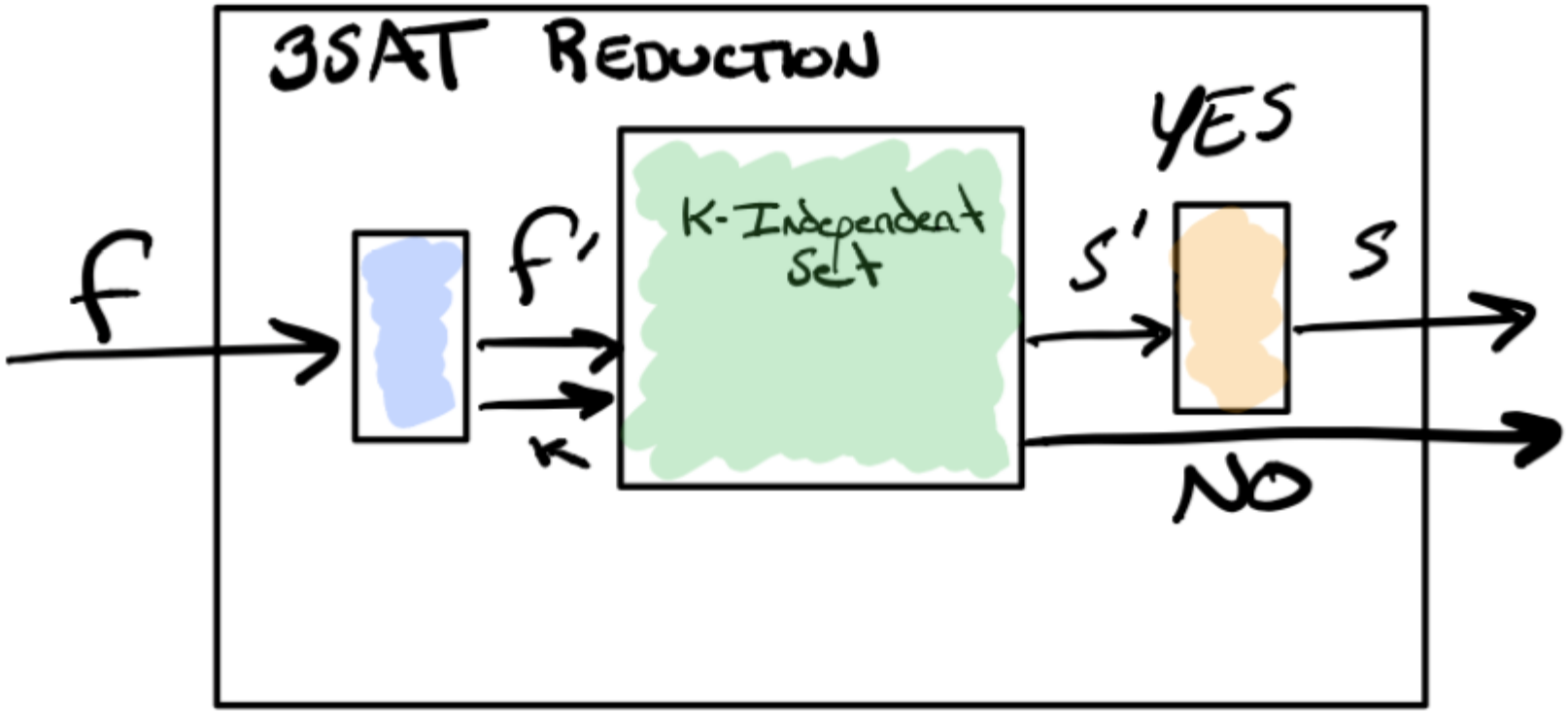
3SAT

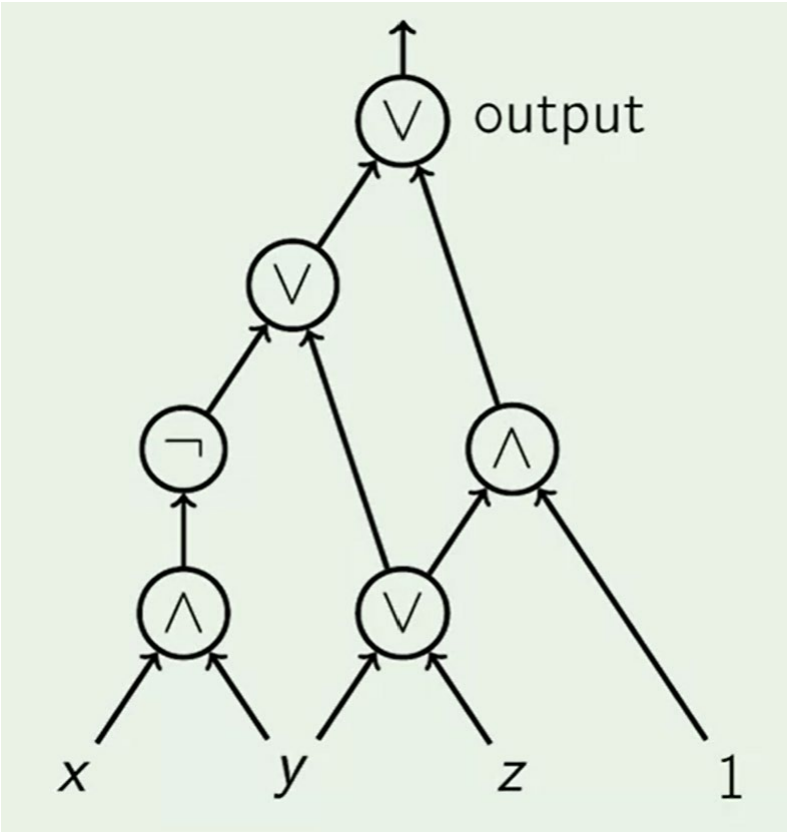
$(x_1 \vee \bar{x}_2 \vee x_3) \wedge$
$(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge$
$(x_4 \vee x_2 \vee x_3)$

$\longrightarrow$

$K = 3$



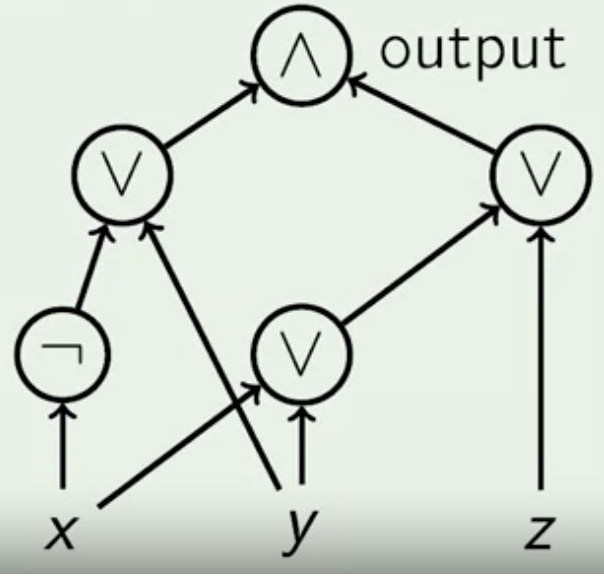

3SAT REDUCTION
f
f'
k
K-Independent Set
S'
YES
S
NO



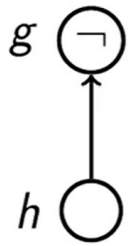
A circuit is a directed acyclic graph of in-degree at most 2. Nodes of in-degree 0 are called inputs and are marked by Boolean variables and constants. Nodes of in-degree 1 and 2 are called gates: gates of in-degree 1 are labeled with NOT, gates of in-degree 2 are labeled with AND or OR. One of the sinks is marked as output.

CNF formula can be represented as a circuit:

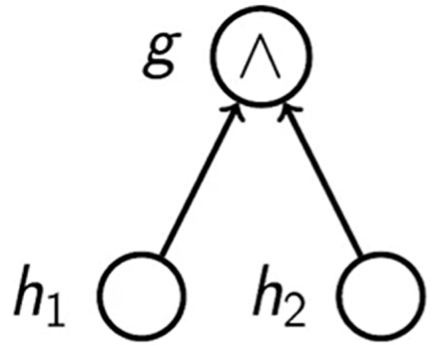
Example: $(x \vee y \vee \overline{z})(y \vee \overline{x})$



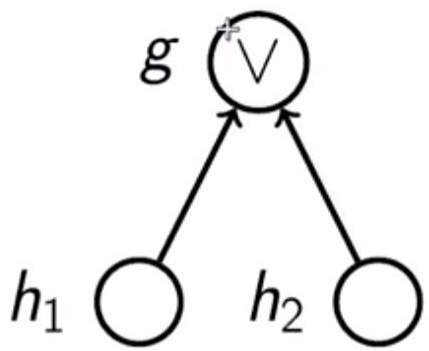
NOT Gates



$$(h \vee g)(\overline{h} \vee \overline{g})$$



$$(h_1 \vee \overline{g})(h_2 \vee \overline{g})(\overline{h}_1 \vee \overline{h}_2 \vee g)$$



$$(\overline{h}_1 \vee g)(\overline{h}_2 \vee g)(h_1 \vee h_2 \vee \overline{g})$$

Cook Levin Theorem

- In computational complexity theory, the Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.

ADA 2023 Tutorial 1

Debarka, Diptapriyo

January 2023

1 Recurrences

Solve the following recurrences: (preferably using recursion trees)

a. $A(n) = 2A(n/4) + \sqrt{n}$

b. $B(n) = B(n/2) + B(n/3) + B(n/6) + n$

c. $C(n) = \sqrt{n}C(\sqrt{n}) + n$

d. $T(n) = T(n/8) + T(3n/5) + 2n$

Solution.

a. Just plug master theorem for $a = 2, b = 4, d = 1/2$, get $O(\sqrt{n}\log n)$.

b. Construct the recurrence tree, and notice that at the total work at each level is exactly n. Why ? First, the work at a node is just the problem size at that node. Now, crucially, the sum of the problem sizes for the different nodes at each level is n. To see this inductively, the problem size at level zero is n. Then, suppose the sum of problem sizes at level k is n for $k \geq 1$. The key observation is that the problem size of any node at level k is exactly the sum of the problem sizes of its children at level $k+1$. This will immediately imply that the sum of the problem sizes at level $k+1$ is the same as that at level k. Now to prove the observation, any node of level k of problem size m has children of problem sizes $m/2, m/3, m/6$ at level $k+1$. Then the sum of these problem sizes of the children $m/2 + m/3 + m/6$ is also m. QED.

   Hence, $T(n) = n \times$ number of levels $= O(n \log n)$

c. Construct the recurrence tree, and notice the same observation as above! The total work at each level is exactly n. First, the work at a node is just the problem size at that node. Now, crucially, the sum of the problem sizes for the different nodes at each level is n. To see this inductively, the problem size at level zero is n. Then, suppose the sum of problem sizes at level k is n for $k \geq 1$. The key observation is that the problem size of any node at level k is exactly the sum of the problem sizes of its children at level $k+1$. This will immediately imply that the sum of the problem sizes at level $k+1$ is the same as that at level k. Now to prove the observation, any node of level k of problem size m has $\sqrt{m}$ children of problem sizes $\sqrt{m}$. Then the sum of these problem sizes of the children $\sqrt{m} \times \sqrt{m}$ is also m. QED.

Hence, $T(n) = n \times$ number of levels. How many levels are there ? To determine this, notice that the problem size at level $k+1$ is the square-root of the problem size at level k. Hence, the problem size at level k is just $n^{1/2^k}$, because the problem size at level 0 is n. To determine the number of levels, just set the problem size to c for some constant c in the base case i.e. solve for k, $n^{1/2^k} = c$. Taking log of both sides, $2^k = \log n / \log c$. Taking log again, $k = O(\log \log n)$.

Hence, $T(n) = O(n \log \log n)$.

d. We use substitution method here. We guess $T(n) \leq 20n$ for all $n \geq 1$.
Base Case: $n = 1$ and $T(1) = 1 \leq 20n$.
Induction Hypothesis: We assume that this guess is correct for all values $1 \leq n \leq k-1$.
Proof for $n = k$.

$$T(k) = T(k/8) + T(3k/5) + 2n$$

Using induction hypothesis we get

$$T(k/8) + T(3k/5) + 2k \leq \frac{5k}{2} + 12k + 2k \leq \frac{33k}{2}$$

Therefore $T(n)$ is $O(n)$.

2 Induction fallacies

Consider the following counting problem : you are hosting a party and you decide to invite n couples. In your party are $2n+1$ chairs arranged in a row, for you and the $2n$ guests. To keep things fun, you want to arrange the $2n+1$ attendees on the chairs such that no couple sits together. Let's call such an arrangement a 'fun' arrangement.

Question: How many fun arrangements are there for the host and the n couples ?

Here's an attempt. *Claim:* The number of fun arrangements of n couples and the host is $(2n)!$.

Let's 'prove' this by induction.

Let $P(n)$ be the proposition that the number of fun arrangements of n couples and the host is $(2n)!$.

Base case: For $n = 1$, the couple must sit on the opposite sides of the host. This can be done in two ways. And indeed, $(2 \times 1)! = 2$.

Induction hypothesis: Let us assume $P(n)$ i.e. n couples and the host can be arranged in $(2n)!$ fun ways.

Induction step: Let us take any fun arrangement for n couples and the host, and try to add an additional couple. There are $2n+1$ people seated already, so we have $2n+2$ seats where we can place the additional couple (since the additional couple must not sit together). Hence, to obtain all possible fun arrangements for the additional couple, it is enough to find all pairs of seats among these possible $2n+2$ seats and multiply that by 2 (since for any two selected seats, the additional couple can be seated in two ways on these two seats). The number of pairs is just $\binom{2n+2}{2}$.

Hence, the number of arrangements of $n+1$ couples is

$$(2n)! \times \binom{2n+2}{2} \times 2 = (2(n+1))!$$

This proves $P(n+1)$.

Is the above proof correct ? Try to take small values of n and try to verify what's going on. If its not correct, what is the mistake?

Solution: The issue with the above argument is the following. The calculation assumes that the *only* way to find a fun arrangement for $n+1$ couple plus host is to start with a fun arrangement for n couple plus the host and then extending it by placing the remaining couple in the gaps. This is a false assumption. Here is the proof. Consider a fun arrangement for $n+1$ couple plus host. Now remove the last couple. However, this *does not* guarantee that you will be left with a fun arrangement for the remaining couple. For instance, it might be the case that one of the person in the final couple was sitting between another couple and upon removing him/her.

In a nutshell, the calculation in the problem shows that the number of fun arrangement for $n+1$ couples is at least the given quantity, but is actually more.

3 n-Stack Game

This game is one-player, to be played by you. You will be given n tiles stacked on each other in a column (so that the height of the stack is n).

The game move for you is as follows : break a stack of height at least 2 into two stacks. Whenever you make this move, you get a score equal to the product of the heights of the two new stacks.

The game ends when all the stacks are of height 1.

Question: Prove that no matter what moves you make over the course of the game, your total score will be the same.

Solution The answer is that no matter what your strategy is at any point in the game, you will end up with the same score $\frac{n(n-1)}{2}$. One can prove this using strong induction. Let $P(n)$ be the following proposition.

$P(n)$: The total score for the game is always $n(n-1)/2$ no matter what the strategy is to break any intermediate stack. Base case is easy. Suppose the hypothesis holds for *every* $k = 1, 2, \cdots k-1$. Now consider $P(k)$. Let the stack be broken at a tiles from below and $b = k - a$. Now, apply strong induction on the remaining stacks of heights a and b respectively to obtain scores $a(a-1)/2$ and $b(b-1)/2$. Finally, we add the score $a \cdot b$ for the first split. This gives us that the final score is

$$a \cdot b + \frac{a(a-1)}{2} + \frac{b(b-1)}{2} = \frac{(a+b)(a+b-1)}{2} = \frac{k(k-1)}{2}$$

ADA 2023 Tutorial 2

February 7, 2023

1 Finding n-th smallest element of two sorted arrays

Suppose you are given two *sorted* arrays A and B, each of size n. Design an $\mathcal{O}(\log n)$ algorithm to find the n-th smallest element of the union of A and B

Solution. We apply a classical divide and conquer approach. Here is the intuition. Suppose we compare the middle elements of the arrays A and B (recall they are sorted and hence this takes constant time). Suppose $A[mid] > B[mid]$ (the other direction is symmetric). Then what do we conclude about the location of the nth smallest element ? Well, it cannot lie in the second half of A, since there are already n elements which are smaller than any element from that half - namely, all elements in the first half of A (including the middle) and all elements in the first half of B). Further, it cannot lie in the first half of B since those elements are strictly less than $A[mid]$ and hence lie among the first $n-1$ elements in the whole array. Hence, we continue our hunt for the n-th smallest element in two subarrays - the first half of A and the second half of B. But wait ! Which element should I look for now ? Well we just look for the $n/2$th smallest element in the combined array of these two subarrays. Why ? Because we already have $n/2$ elements that lie among the first $n-1$ elements of the combined array. So it is enough to find the $n/2$th smallest element in the remaining search space.

2 Finding Majority

You are given an array of n elements, not necessarily distinct. An element is called *majority* if it appears strictly more than $n/2$ times. Your task is to find a majority if it exists. Importantly, the only operation you are allowed to perform to compare two elements is to test if the two elements are identical or different, i.e. there is no notion of an element being less than or greater than another element (and hence *you cannot sort*!), For instance, say the elements are DNA samples, and you can only test if two samples match, there is no notion of ordering, $<$ or $>$ on the samples.

a. Design a *divide-and-conquer* algorithm to find the majority element or report there exists none. Your algorithm should perform $\mathcal{O}(n \log n)$ tests.

   Solution. The following is the recursive algorithm at a high level.

   (a) If A has 3 or less elements, just return majority by counting. Else do the following.

   (b) Divide the given array A at the middle - call the left half L and the right half R.

(c) Recursively find the majority in L - call it ℓ, null if no majority and majority in R - call it r - , null otherwise.

(d) If null returned in both recursive calls, return 'no majority'

(e) Else do the following. Count the number of times ℓ appears in R. If the sum of count from the recursive call on L and this count exceeds $n/2$, return *ell* and the total count. Else repeat the analogous thing for r.

(f) If none of the counts above exceeds $n/2$ return null.

The combine step clearly runs in $\Theta(n)$ time since all we are doing is going over each half linearly and comparing with a fixed element. Hence, the recursion is $T(n) = 2T(n/2) + \mathcal{O}(n)$ which solves to $\mathcal{O}(n \log n)$

Correctness.

Claim 1 *The above algorithm returns the majority in array A and its count correctly if and only if such such an element exists.*

Proof: We prove this by induction on the size of the array. The base cases are easy to check. Now suppose claim holds for all arrays of size $1, 2, \cdots n-1$. Now consider A which is of size n. By induction hypothesis, we know that the algorithm returns correct answer for the subarrays L and R. Now we prove the two directions.

- Suppose there exists a majority element x in A. Then, by Pigeon Hole Principle, either L or R must contains strictly more than $n/4 + 1$ copies of x. Hence, the algorithm correctly enters step (d). Now it is easy to see that the algorithm will return the correct answer for A.
- Suppose there is no majority in A. Now there could be two cases. If null is returned for both calls in step (c), then we are done by induction hypothesis. Suppose that does not happen and algorithm enters step (d). (Note: This can happen. There might be a majority element in either L or R even if there is no majority in A). In this case, both the counts has to return less than $n/2 + 1$. Again, note that we are implicitly using induction hypothesis on arrays L and R of size $n/2$ each when we are assuming that they return the majority count correctly.

□

b. * Do the above in $\mathcal{O}(n)$ tests. **Solution**

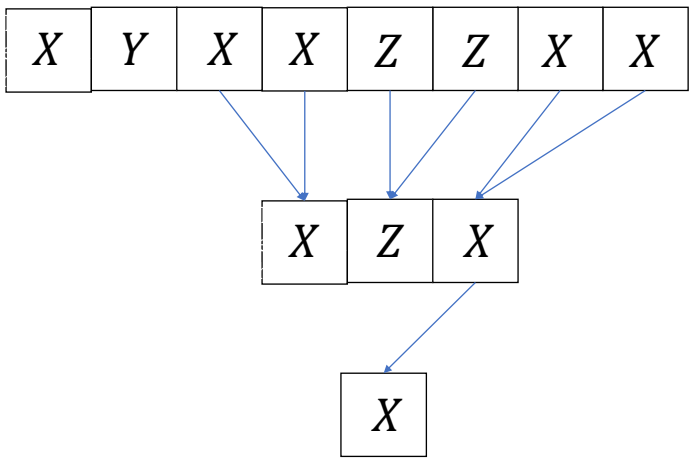
There are two ways that I know of for doing this. I am showing you one of them which I feel uses the spirit of divide and conquer, although its not the usual divide and conquer thing. I am assuming n is a power of 2 for simplicity.

(a) If A contains one element return this element as majority. Else do the following

(b) Keep making disjoint pairs by testing adjacent elements. If they are different, remove them from A. Otherwise replace both copies with a single copy of the common element. In case the array length is odd, just copy the last element as it is.

(c) Recurse on the modified array.

(d) Suppose the above recursion returns x. Now make a final pass through the original array A to check if x is indeed the majority. (This is a super important step !!!!)



See figure below for an example run.

Let us introduce the notation A_ℓ to denote the array after ℓ recursive calls. For example, $A_0 = A$.

Runtime: Its clear that the runtime for level ℓ is $\mathcal{O}(|A_\ell) = \mathcal{O}(n/2^\ell)$ since all we are doing is a linear scan through the array and comparing adjacent elements. Note that at every step, you are reducing the size of the array by at least half and hence $|A_\ell| \leq n/2^\ell$. So, the total runtime is $\sum_{\ell=0}^{\log n} c \cdots \frac{n}{2^\ell} \leq 2cn = \mathcal{O}(n)$. You need another $O(n)$ for the final pass.

Correctness

Claim 2 *An element x survives in $A_{\log n}$ if x is a majority in A.*

Proof: Suppose x is a majority in A. We show by induction on ℓ that x is a majority in $A_\ell, \forall \ell = 0, 1, 2, \cdots \log n - 1$. The base case is trivially true. Now consider a level $\ell + 1$ and assume by induction hypothesis that x is majority in A_ℓ. Suppose for the sake of contradiction x is not so in $A_{\ell+1}$. Now, let us count the number of occurrences of x in A_ℓ. Well, each occurrence of x can account for two occurrences of x in A_ℓ. On the other hand, each deleted pair from A_ℓ accounts for at most 1 occurrence of x. Thus, x cannot be a majority in A_ℓ leading to a contradiction. (Note the subtle points where we are using that pairs are disjoint - do you see where ?).

□

Note the very subtle point here. The above proof shows that the recursive part of the algorithm returns a majority element correctly *if* such an element exists. However, it might return a wrong element as majority in case there is no majority element!. This is why you need a final pass in order to check whether the returned element is indeed majority or not. As an exercise, find an example where the recursive part can spit out a wrong answer in case there is no majority.

3 Local minimum in an array

1.* Given an array A, we say that an element $A[i]$ is a local minimum if $A[i] \leq A[i+1]$ and $A[i] \leq A[i-1]$ (we only check the inequality for those elements $A[i+1], A[i-1]$ which exist). In other words, a local minimum is an element which is less than or equal to each of its (at most 2) neighbors. Give an algorithm to find *any* local minimum in an array of n elements by making $\mathcal{O}(\log n)$ comparisons.

Solution. First notice that a local minimum always exists - the global minimum (i.e. the smallest element) is always a local minimum. Naively finding it would need $O(n)$ comparisons, but we can do better since we only need *any* local minimum, not necessarily the global minimum. The idea, as one might guess, is to use binary search.

Consider the middle element $A[n/2]$ (assume n is a power of 2). If $A[n/2]$ is a local minimum, just return it. Else it partitions the array into two halves L, R. But the question is - *in which half can we be sure to find a local minimum ?*

The idea is to decide locally i.e. if the left (resp. right) neighbor of $A[n]$ is smaller than $A[n]$, then we should look in the left (resp. right) subarray. More formally, consider algorithm 1.

Algorithm 1 Algorithm for local min in an array

```
procedure LocalMin(A)                                   ▷ Local Min of array of length n
    If A has size at most 3, find a local minimum by brute force. Otherwise,
    if A[n/2] ≤ A[n/2 − 1] and A[n/2] ≤ A[n/2 + 1] then return n/2
    else if A[n/2] > A[n/2 − 1] then                    ▷ Look for a local min on the left
        L ← A[1] ... A[n/2]
        return LocalMin(L)
    else                                                ▷ Look for a local min on the right
        R ← A[n/2] ... A[n]
        return LocalMin(R)
    end if
```

It is easy to see that the algorithm 1 makes only $\mathcal{O}(\log n)$ comparisons, similar to binary search, the recurrence is $T(n) = T(n/2) + O(1)$.

Let us see why it will always find a local minimum. Consider the case when $A[n/2]$ is not a local minimum. Then, the algorithm works because the local min x from the recursive call will not be $A[n/2]$. This is because we choose the half to recurse on so that this happens. Hence, x will have the same neighbors in the subarray recursed on, as its neighbors in A. Hence, x will also be a local minimum in A.

Formal proof by induction:

$P(n)$: the algorithm 1 finds a local minimum for any array of size n.

Base case: $P(1), P(2), P(3)$ hold - If A has at most 3 elements, we are done by brute force.

Induction hypothesis: Suppose $P(k)$ holds whenever $k < n$.

Induction Step: If $A[n/2]$ is a local minimum, the algorithm will return it. Otherwise, either $A[n/2] > A[n/2-1]$ or $A[n/2] > A[n/2+1]$. Let us focus on the first case (the other case is symmetric). Then, by applying the induction hypothesis on the length of L, the algorithm returns a local min x in L. x cannot be $A[n/2]$ (because $A[n/2]$ is bigger than its left neighbor

in L). Since x is not the rightmost element of L, the neighbors of x in L are the same as the neighbors of x in A. Hence, x is also a local min in A.

2.** Given an $m \times n$ 2-D array A, we say that an element $A[i][j]$ is a local minimum if it is no more than each of its (at most four) neighbors i.e. $A[i][j] \leq A[i+1][j]$, $A[i][j] \leq A[i][j+1]$, $A[i][j] \leq A[i-1][j]$ and $A[i][j] \leq A[i][j-1]$ (again, we only check the inequality for those elements which exist). Give an algorithm to find any local minimum in A by making $\mathcal{O}(n \log m)$ comparisons.

Solution. The following observation is the key to everything:

Lemma 3 *Consider the array B of length n defined as $B[j] = \min_i A[i][j]$. Then, a local min in B is a local min in A.*

Proof: Suppose $B[j]$ is a local minimum in B. Now by definition of $B[j]$, $A[i][j] \leq A[i+1][j]$ and $A[i][j] \leq A[i-1][j]$. Also, $A[i][j] = B[j] \leq B[j-1] \leq A[i][j-1]$ and $A[i][j] = B[j] \leq B[j+1] \leq A[i][j+1]$, using that $B[j]$ is a local minimum. Hence, $A[i][j]$ is a local minimum in A. □

Now, we can just find a local min in B by using the algorithm 1 for finding a local min in a 1D array. We do not need to find all the entries of B, whenever the algorithm 1 needs an entry $B[j]$ of B, just find the smallest entry of the jth column of A in m comparisons. Since algorithm 1 needs $\mathcal{O}(\log n)$ comparisons of entries of B, the total number of comparisons is $\mathcal{O}(m \log n)$.

This can actually be improved even to $\mathcal{O}(\log m + \log n)$!)

ADA 2023 Tutorial 3

Diptapriyo, Debarka

February 2023

This tutorial is more warmup on DPs. What we would like you to do for each of the problems is:-

1. Define the subproblems clearly
2. Write a recursion using the above definition and argue properly about why the recursion is correct (this is the optimal substructure property)
3. Implement using tables and argue runtime.

1 Maximizing Revenue by choosing jobs in different weeks

Suppose that there are n weeks, a plan is specified by a 'low stress job' and a 'high stress job' or 'none' in each of the weeks. For any $i \geq 2$, if a high stress job is chosen for the week i, then no job can be chosen in the week $i-1$. However, if a low stress job is chosen for the week i, then both high stress and low stress job can be chosen for the week $i-1$. In the week 1, a high stress job and low stress job both can be chosen. A low stress job gives revenue a_i and a high stress job gives a revenue of b_i. Give an algorithm that provides a plan for n consecutive weeks that maximizes the revenue.

Solution: If $n = 0$, then the scenario is obvious that the total revenue is 0. Suppose that $n = 1$. Then, both low stress job and high stress job can be chosen or no job can be chosen at all. So, $\max(a_1, b_1)$ is the solution when $n = 1$. For $n \geq 2$, the main trick here is to consider two cases. First case is that at a *low stress job* is chosen at the week i, and the second case is that a *high stress job* is chosen in week i. The first case gives a_i revenue in the i'th week and the second case gives b_i revenue in the i'th week. For the first case, the $i-1$'th week can have any job. On the other hand, for the second case, the $i-1$'th week cannot have any job but the $i-2$'th week can have any job.

So, we consider the subproblem REVENUE(i) as the optimal (maximum) revenue for the weeks $\{1, 2, \ldots, i\}$.

- **Base Cases:** REVENUE$(0) = 0$ and REVENUE$(1) = \max(a_1, b_1)$.
- **Recurrence:** REVENUE$(i) = \max\left(\text{REVENUE}(i-1) + a_i, \text{REVENUE}(i-2) + b_i\right)$.
- **Final solution:** REVENUE(n) gives the final solution.

The rest is converting this recurrence relation into an iterative algorithm by constructing the table.

- Initialize REVENUE$(0) \leftarrow 0$ and REVENUE$(1) \leftarrow \max(a_1, b_1)$.
- For $i = 2, 3, \ldots, n$ in this order set
  REVENUE$(i) \leftarrow \max\left(\text{REVENUE}(i-1) + a_i, \text{REVENUE}(i-2) + b_i\right)$.
- Output REVENUE(n).

Correctness Proof: By induction on the number of weeks, i.e. n.

2 Maximum Switching Subarray

Suppose you are given an array of distinct real numbers, $A[1:n]$. Define a switching subsequence of A as a sequence $A[k_1], A[k_2], \cdots A[k_\ell]$ such that ,

$$A[k_i] < A[k_{i+1}], \text{for odd i}$$
$$A[k_i] > A[k_{i+1}], \text{for even i}$$

In plain English, the sequence switches between increasing and decreasing, starting with increasing. The goal is to find the longest switching sub-sequence of A. Deisgn a linear time algorithm for this.

Solution. The main trick here is to realize that you actually need to define two different subproblems for each $i = 1, 2, \cdots n$. The rest is very similar to the above problem with a minor twist.

Suppose $longestInc(i)$ denote the longest switching subsequence ending at i where the last entry is bigger than the preceding one while $longestDec(i)$ denote the longest switching subsequence ending at i where the last entry is smaller than the preceding one. Then the following is true :

$$longestInc(i) = max_{j<i, A[j]<A[i]} longestDec(i) + 1$$
$$longestDec(i) = max_{j<i, A[j]>A[i]} longestInc(i) + 1$$

Now the rest is again converting this to an iterative solution using table.

(Note: There is a simpler $\mathcal{O}(n^2)$-time DP solution to solve this problem. Can you figure that out?)

ADA 2023 Tutorial 4

Debarka, Diptapriyo

1. Define the subproblems clearly
2. Write a recursion using the above definition
3. Argue properly about why the recursion is correct (this is the optimal substructure property)
4. Implement using tables and argue runtime.

1 Inventory Management

Consider the following inventory problem. You are running a store that sells some large product (let us assume you sell trucks), and predictions tell you the quantity of sales to expect over the next n months. Let d_i denote the number of sales you expect in month i. We will assume that all sales happen at the beginning of the month, and trucks that are not sold are stored until the beginning of the next month. You can store at most S trucks, and it costs C to store a single truck for a month. You receive shipments of trucks by placing orders for them, and there is a fixed ordering fee of K each time you place an order (regardless of the number of trucks you order). You start out with no trucks. The problem is to design an algorithm that decides how to place orders so that you satisfy all the demands d_i, and minimize the costs. In summary: There are two parts to the cost.

- First, storage: it costs C for every truck on hand that is not needed that month. Second, ordering fees: it costs K for every order placed.
- Each month, you need enough trucks to satisfy the demand d_i, but the amount left over after satisfying the demand for the month should not exceed the inventory limit S.

Solution.

Subproblems. The subproblems are a bit subtle here. We define $\mathsf{OPT}(i, s)$ to be the optimal solution for months i to n given that there are s trucks at the beginning of month i (before placing any order).

Recurrence. Idea : Look at $\mathsf{OPT}(i, s)$. Now there could be two cases regarding what happens in month i

1. Suppose $s > d_i$. This is the easier case, in this case we do not order any new truck in month i, pay a storage cost of $C(s - d_i)$. The residual solution would be (at least we hope) $\mathsf{OPT}(i + 1, s - d_i)$.

2. Suppose $s \leq d_i$. In this case we need to order new trucks - note that this cost is just K irrespective of the number of trucks ordered. Now suppose you have $0 \leq s' \leq S$ trucks remaining after meeting the demand. Then you pay Cs' for the storage. The remaining solution (we hope) would be $\mathsf{OPT}(i + 1, s')$. Hence, the recurrence is :

$$\mathsf{OPT}(i, s) = \min \begin{cases} \mathsf{OPT}(i + 1, s - d_i) + C(s - d_i) & \text{if } s \leq d_i \\ \min_{0 \leq s' \leq S} K + Cs' + \mathsf{OPT}(i + 1, s') & \text{if } s' > d_i \end{cases}$$

What is the base case ?

Correctness. We are going to prove the correctness of the recurrence in two different case.

1. $(s \leq d_i)$: We claim that in this case $\mathsf{OPT}(i + 1, s - d_i) + C(s - d_i) = \mathsf{OPT}(i, s)$. To prove this, let us take an optimal solution for the months i to n and s number of trucks being available at the start of month i. Now let us consider this schedule and look at the beginning of month $i + 1$. Clearly we remain with $s - d_i$ trucks at the beginning of month $i + 1$. Hence, the residual schedule is clearly a feasible solution for $i + 1$ to n months when there are $s - d_i$ trucks at the beginning of month $i + 1$ and the cost of this solution is $\mathsf{OPT}(i, s) - C(s - d_i)$. Now suppose for contradiction this residual solution is not $\mathsf{OPT}(i + 1, s - d_i)$. Hence $\mathsf{OPT}(i + 1, s - d_i) < \mathsf{OPT}(i, s) - C(s - d_i)$. Now consider a schedule where you sell d_i trucks in month i and the rest of the schedule is $\mathsf{OPT}(i + 1, s - d_i)$. This schedule is a feasible schedule for months i through n when we start with s trucks. But it has cost

$\mathsf{OPT}(i+1, s-d_i) + C(s-d_i) < \mathsf{OPT}(i,s)$ contradicting the optimality of $\mathsf{OPT}(i,s)$.

2. $(s > d_i)$: The proof is very similar. **Please figure it out yourself. You need to practice writing proofs**

Algorithm. Quite straightforward.

2 Knapsack cover

You are given items i with sizes w_i and values v_i, and a target value V. Find a subset of items of minimum total size with total value at least V (or decide that there is no such subset).

Solution.

Subproblem.

For $1 \leq i \leq n, 0 \leq P \leq V$,

$B(i,P)$: minimum total size of items $1, 2, \ldots, i$ whose value is at least P

Recurrence.

$$B(i,P) = \min \begin{cases} B(i-1,P) \\ B(i-1,P-v_i) + w_i \quad \text{if } P \geq v_i \end{cases}$$

Base case:

$$B(0,P) = \begin{cases} 0 \quad \text{for } P = 0 \\ \infty \quad \text{otherwise} \end{cases}$$

Correctness.

Consider two cases about the optimum solution OPT for $B(i,P)$.

1. $i \in OPT$**:** We show that $OPT \setminus i$ is optimal for the subproblem for $(i-1, P-v_i)$. It has value $P - v_i$ and uses items $1, 2, \ldots, i-1$ and hence is feasible. To show that it has the optimal size, suppose for contradiction that, there is a solution S of smaller total size than $OPT \setminus i$.

Then, $S \cup i$ is a solution of value at least $P - v_i + v_i = P$ using items $1, 2, 3, \ldots i$ and hence is feasible for the subproblem for (i, P). But then the total size of $S \cup i$ is $\text{size}(S) + w_i < \text{size}(OPT \setminus i) + w_i = \text{size}(OPT)$; which contradictions the definition of OPT.
Hence, $OPT \setminus i$ has size $B(i-1, P - v_i)$ and OPT has size $B(i-1, P - v_i) + w_i$.

2. $i \notin OPT$**:** Write yourself, similar to the above.

Pseudocode.

Write yourself.

Time Complexity.

There are nV subproblems and each takes $O(1)$ time. Hence, the time complexity is $O(nV)$.

Algorithm Design and Analysis
CSE222 Winter '20

Tutorial 7

Problem 1 Given a directed graph $G(V, E)$, the $\text{Rev}(G) = G_r(V, E_r)$ is defined as follows. Every $(u, v) \in E$ if and only if $(v, u) \in E_r$. Informally, the G_r is defined as the directed graph G_r such that all the edges of G are in the opposite direction in G_r Design a linear time, i.e., $O(V + E)$ times algorithm to compute G_r. The algorithm should be able to output the adjacency list of of G_r.
Note to TAs: This is a warm-up exercise to get familiar with adjacency list data structure.
Solution. Assume that G is represented in adjacency list format and $Adj[u]$ is the linked list containing all the edges going out of u in G. Initialize an adjacency list $Adj_r[u] =$ for all $u \in V(G)$. For all $u \in V$, if $v \in Adj[u]$, then insert u at $Adj_r[v]$.

```
for all u ∈ V do
    Initialize Adj_r[u] ← ∅;
end
for all u ∈ V do
    for all v ∈ Adj[u] do
        Insert u into Adj_r(v);
    end
end
Output the vertices V with Adj_r[v] for all v ∈ V;
```

Note to TAs: The solutions for both the following can be found in this link

Problem 2 A vertex in an undirected graph is a cut vertex if its removal (along with any incident edges) causes the graph to become disconnected. In this problem we will (at a high level) adapt the algorithm for computing bridges to compute cut vertices, by describing the conditions under which a vertex is a cut vertex. Let $G = (V; E)$ be a connected undirected graph, and suppose that you run DFS on G and (as in Lecture) you compute the value of $start[u]$ (i.e. the starting timestamp of u) for every vertex $u \in V$. Prove each of the following assertions:
(**Note to TAs:** Do not tell them the following upfront. Rather ask them to come up with what properties are required to design the algorithm. They should be able to come up with all these three things)

(a) The root of the DFS tree is a cut vertex if and only if it has two or more children.

(b) No leaf of the DFS tree can be a cut vertex.

(c) A non-root, internal vertex u of the DFS tree is a cut vertex if and only if it has a child v such that there is no back edge of the formß (x, y) such that $x = v$ or x is a descendant of v but y is an ancestor of u.

(d) Let $LOW(v)$ is defined as follows.

$$= \min\{start(v), start(w) : (u, w) \text{ is a back edge for some descendant } u \text{ of } v\}$$

Show how you will use the ideas of DFS to show how to compute $LOW(v)$ for all $v \in V(G)$ in $O(|V| + |E|)$-time.

Algorithm Design and Analysis
CSE222 Winter '23

Tutorial 8

The following two properties of MSTs and Graph Traversals would be required for most of the problems below.

i. *Cut Property.* For any edge e, if there exists a cut $S, V \setminus S$ such that e is the cheapest edge (or one of the cheapest edges in case edge costs are not distinct) crossing the cut, then e is necessarily part of the MST of G. This was proved in the class.

ii. *Cycle Property.* If any edge e in G is the most expensive edge in *some cycle* in G, then e cannot be in the MST of G.

Why is this true ? Suppose not - that is e is the most expensive edge of some cycle C and yet it is in the MST. Now consider the cut formed by the connected components upon removing e from the MST. By the Double Cross Lemma (done in class), there has to be another e' that is crossing this cut in G and belongs to the cycle C. Remove e from and add e' to the MST. Clearly this produces another spanning tree - why ? Because each side of the cut forms connected components even after removing e and hence adding e' connects everybody. Further, it is cheaper than the MST - hence a contradiction.

Problem 0 Prove the cycle property.

Problem 1 Given a connected graph $G = (V, E)$ with costs c_e on the edges and a spanning tree $T \subset E$, the *bottleneck* of T is the cost of the most expensive edge in T. The following problem is about finding the minimum bottleneck spanning tree (MBST) of G. For each of the following, either prove the statement or show a counter-example.

a. The MST of G is always an MBST of G.

Solution. Let MST be $T^\star$ and MBST be $\hat{T}$. Suppose the above is not true for the sake of contradiction. Hence the most expensive edge in $T^\star$, say e, is not part of the $\hat{T}$ and further, all the edges in $\hat{T}$ are cheaper than e. Thus, $\hat{T} \cup \{e\}$ has a cycle for which e is the most expensive edge. But then by Cycle property, e cannot be part of $T^\star$ leading to a contradiction.

b. An MBST of G is not necessarily an MST of G

Solution. Easy examples, try to find yourself.

Give a $O(|E| \log |V|)$ time algorithm to find a spanning tree whose bottleneck is minimum.

Problem 2 Suppose you are given a connected graph $G = (V, E)$ with costs c_e on the edges and a *minimum* spanning tree $T \subset E$. Give an $O(|V| + |E|)$ algorithm to find a minimum spanning tree when

a) The cost of a given edge e is decreased.

Solution sketch.

- Case I : If e is part of MST, then the old MST is still the MST. To see this, recall the cut property of edges and MSTs of a graph as done in the lecture. The converse of this is also true. That is, any edge in MST is necessarily one of the cheapest edges crossing some cut - why ? This follows from Prim's algorithm's correctness since every time it adds an edge to the tree, it is indeed (one of) the cheapest edges crossing some cut. Here e is such an edge. Now if the cost of e decreases, it clearly still remains the cheapest edge crossing the corresponding cut. Hence, the MST will remain unchanged.
- Case II : If e is not part of MST, then find the unique fundamental cycle formed by MST union e. Remove the largest cost edge from that cycle. All these can be done using bfs/dfs etc. The correctness follows from cycle property which states - The largest cost edge of any cycle in the graph G is not part of the MST.

b) The cost of a given edge e is increased.

- If e is not part of the MST, it can not become part of new MST
- If e was part of MST - First let us understand what needs to be done. Suppose you remove the edge e from MST. Note that the two components that are formed as a result essentially form the cut which certifies the existence of e in the MST (by the Cut Property and Prim's Algorithm). Now all you need to do is choose the minimum cost edge that is crossing this cut and plug it back (note that this could be the edge e itself). This can be done in linear time as follows. Firstly, the connected components can be be found by running two BFS/DFS - one from each of the endpoints of e. Then just find the minimum edge crossing this cut and add it to T - clearly this can be done in $O(E)$ time.

Problem 3 Suppose that an n-vertex graph $G = (V, E)$ contains two vertices s and t such that the distance between them is strictly greater than $\frac{n}{2}$.

(a) Prove that there exists a vertex u that is neither s nor t such that deleting u destroys all the s-t paths in G.

(b) Give an $O(m + n)$-time algorithm to find such a vertex v.

Solution:

(a) The proof idea is based on a counting argument. We have to exploit the property of BFS carefully. If the BFS algorithm is started from the vertex s, then it computes a value $Layer(v)$ (as discussed in the lecture) for all the vertices $v \in V$. This value $Layer(v)$ contains the distance from s to v, i.e. the number of edges in a shortest path from s to v in G. Note that $Layer(s) = 0$ and $Layer(t) > n/2$. Let us now consider the vertices that occur with Layer-values $1, 2, \ldots, \lfloor n/2 \rfloor$ of the BFS tree T_s starting from s, i.e. all the vertices that are of distance at most $n/2$ from s. At least one such vertex has to be deleted in order to destroy all s-t paths. Suppose for the sake of contradiction that there is no vertex u such that deleting u destroys all the s-t paths in G.
Case 1: n is odd. Since $\lfloor n/2 \rfloor + 1 = \lceil n/2 \rceil$, it holds that $Layer(t) \geq \lceil n/2 \rceil$. Moreover, $2\lfloor n/2 \rfloor = n - 1$. By our assumption, for every $1 \leq i \leq \lfloor n/2 \rfloor$, there are 2 vertices x, y in

the BFS-tree T_s such that $Layer(x) = Layer(y) = i$. Then, the total number of vertices that occur with Layer-values $1, 2, \ldots, \lfloor n/2 \rfloor$ of the BFS tree T_s is at least $n/2-1$ excluding the vertices s and t. This contradicts that the total number of vertices is n.

Case 2: n is even. The proof idea is similar as before.

(b) Using the proof idea above the algorithm is simple. The main steps are the following.

- Invoke $BFS(G, s)$ starting from s and compute an array contaning the $Layer(v)$ values for all $v \in V$.
- For every $1 \leq i \leq \lfloor n/2 \rfloor$ check if only one vertex $u \in V \setminus \{s, t\}$ satisfies $Layer(u) = i$. This can be done in $O(n)$-time by putting a counter array. As soon as such a vertex u is found, return that vertex u.

Note that this entire process takes $O(n + m)$-time.

Algorithm Design and Analysis
CSE222 Winter '23

Tutorial 9

This tutorial involves problem discussions on problems related to BFS and DFS of graphs.

Problem 1 We want linear time, that is $O(m+n)$ algorithms. No credit for anything that is slower.
Hint 1: The distance array D has nothing to do with the edges in the graph.
Hint 2 : DP !
Solution. The goal of this problem is compute the length of a longest path from s to v such that $(s,v) \in E(G)$, and the longest path is of the form $s, v_1, v_2, \ldots, v_k (= v)$ such that $D[v_i] \geq D[v_{i+1}]$.

Approach 1 (sketch): Construct an auxiliary directed acyclic graph $D_G = (V, A)$ as follows. For every neighbor u of s, make (s,u) a directed edge. Then, for every u and v, if and only if $D[v] > D[u]$, direct the edge uv from $utov$. (intuitively, this implies that Judy can teleport from u to v as per the given rule).

Observe that the resulting digraph is acyclic. To see this, assume there is a cycle $v_0, v_1, v_2, \cdots v_k, v_0$. But then $D[v_0] < D[v_1] < \cdots < D[v_k] < D[v_0]$ which is clearly a contradiction.

Subproblems: For every vertex v, we denote $OPT(v)$ to be the longest path from s to v in D_G.

Recurrence

$$OPT(v) = 1 + \max_{u:(u,v)\in A} OPT(u)$$

. Also, $OPT(s) = 0$.

The subproblem that solves the actual problem. $\max_{u:(s,u)\in A} OPT(u)$. This is because the path has to end at one of the neighbors of s so that Judy can take a last hop back home as given in the problem.

Optimal Substructure Property: (not required for credit) Let P be an optimal path from s to v in D. Clearly, the sequence of vertices in P represents a path in G also. Moreover, by construction of D_G, the path P gives a non-increasing sequence of vertices (except for s) in G (with respect to the values in D-array). Let u be the predecessor of v in P. Then, we claim that the subpath P_u from s to u is an optimal solution for $OPT(u)$.

Justification: Suppose that this is not the case. Then, let some other path P'_u is an optimal path from s to u. Then, consider the path P' obtained by appending v at the end of P'_u. As P'_u has more vertices than P_u and it satisfies the criteria of non-increasing D-values, the path P' also satisfies the same. Clearly, P' has more vertices than P. Hence, P is not an optimal path from s to v.

Algorithm: We are now ready to describe an iterative algorithm.

- Construct D_G as described above.
- Initialize $GM[s] = 0$ and for all other vertices $v \in V(G) \setminus \{s\}$, $GM[v] = -\infty$.
- Run a topological sort on D_G. Clearly s is the source of this topological ordering $(s, u_1, u_2, \ldots, u_{n-1})$.
- For every $1 \leq i \leq n-1$, $GM[u_i] = 1 + \max_{j<i:(u_j,u_i)\in A} GM[u_j]$.

- For every $u \in N_G(s)$, output $\max\{GM[u]\}$.

Running time justification: Observe that in this algorithm, every edge of G is checked only once while computing the values of $GM[u]$ for all $u \in V(G)$. Hence, the running time of the algorithm is $O(m+n)$.

Approach 2 (memoization based): We construct a DAG in the same way as before. In addition, we define the subproblems and the optimal substructure in the same way as before.

Algorithm: Construct a DAG $D_G = (V, A)$ as described above. For all $v \in V(G) \setminus \{s\}$, initialize $GM[v] = -\infty$ and $GM[s] = 0$. For every $v \in V(G) \setminus \{s\}$ in any order, invoke $COMPUTE(v)$ as follows.

a) If $GM[v] \neq -\infty$, then return $GM[v]$.

b) Else,

$$q = 1 + \max_{u:(u,v)\in A} COMPUTE(u)$$

.

c) Assign $GM[v] = q$ and Return $GM[v]$.

Finally output $\max_{u:(s,u)\in A} GM[u]$.

Running time:

We first claim that $GM[v] \neq -\infty$ when $COMPUTE(v)$ is called the second time. Suppose this is not the case for the sake of contradiction. This means that Step(3) was not executed before the second call to $COMPUTE(v)$. Let us trace back the recursive calls to $COMPUTE$ starting with the second recursive call to v. Then by construction of the DAG, we shall have a sequence of vertices $v = v_0, v_1, v_2, \cdots v_k = v$ such that there is an edge in A from v_{i+1} to $v_i, \forall i = 0, 1, \cdots, k-1$. This means we have a directed cycle in the DAG leading to a contradiction.

Hence, for every recursive call to v starting with the second one, only Step (1) is executed. The total number of times this can happen is $|outdegree(v)|$. Hence the total number of times Step(1) is executed is at most $\sum_{v\in A} |outdegree(v)| \leq m$. The total number of times steps(2) and (3) are executed for a fixed vertex v is at most one by the first claim. Hence the running time is $O(m+n)$.

Problem 2 You are given a graph $G = (V, E)$ where the weights of the edges can have only three possible values - 2, 5 and 7. Design a linear time , that is $(E+V)$ time algorithm to find the minimum spanning tree of G. *Hint:* Modify a known algorithm. No proof of correctness required. But argue runtime for credit.

Approach 1:

We can use Prim's algorithm. Prim's algorithm can be implemented in time $(|E|t + |V|)$ where t is the time taken to insert or delete an element in a priority queue of size at most $|E|$.

In class, we used a min-heap to implement the priority queue, which gave a time complexity of $(|E| \log |V|)$. However, if the element values can have only a small number (three in this case) of possibilities, we can implement the priority queue to support insertions and deletions in $O(1)$ time.

Priority queue implementation: maintain 3 lists of elements - each for one possible value of the elements. For insertion, add the element to the list for the corresponding value.

To delete the element with smallest value, return (and delete) an arbitrary element from the list of smallest value which is non-empty.

Time complexity of priority queue: If there are k possible values of the elements, the priority queue insertion and deletion take $O(k)$ time each, since insertion and deletion in a list takes $O(1)$ time.

Running time of Prim's implementation: Hence, the running time of Prim's algorithm for k values of edge weights is $O(k|E| + |V|)$ which is $O(|E| + |V|)$ for $k = 3$ as in the question.

We now state and analyze Prim's algorithm for completeness.

Prim's algorithm: Maintain a set X of vertices reachable from a source s using the tree edges. Also maintain a priority queue Q of edges with the invariant that every edge in the cut $(X, V - X)$ is in Q, with value equal to the edge's weight. Initialize Q with the edges incident to s.

At every step, find the cheapest edge crossing the cut $(X, V - X)$ and add it to the tree (and update X). To implement this, repeatedly find (and delete) the cheapest edge (u, v) in Q. If (u, v) crosses the cut $(X, V - X)$, (say with $u \in X$ and $v \notin X$), add (u, v) to the tree, add v to X and insert into Q the edges incident to v.

Analysis of Prim's algorithm: Prim's algorithm is correct due to the cut-lemma which says that the cheapest edge crossing any cut $(A, V - A)$ is in a minimum spanning tree.

The running time is dominated by the number of insertions and deletions in the priority queue. Each edge (u, v) can be inserted into the priority queue at most twice, when either of u or v enters the set X. The number of deletions cannot exceed the number of insertions. Hence, the number of priority queue operations is at most $4|E|$. Hence, the time complexity is $(t|E|)$ where t is the time to insert or delete a single element in the priority queue.

Approach 2. Another solution could be modifying Kruskal's algorithm. Let us recall the algorithm first. Sort the list of edges in non-decreasing order of weights. Maintain the connected components of set of edges F picked so far. Start with n trivial components - each a singleton vertex. At any iteration, the algorithm considers the next edge (u, v) in the sorted list. If u and v belong to different components, add (u, v) to the set F and merge the two components of u and of v. Otherwise, discard the edge (u, v). It can be shown that at the end there can be at most one component in F (provided the given graph is connected) and there are no cycles in F. Further, due to the cycle property of MSTs (or equivalently, the cut property) and the fact that we consider edges in non-decreasing order, F is indeed an MST.

Realize that you can implement Kruskal faster. Now, the runtime of this procedure is $(E \log V)$. There are two potential sources of the logarithmic factor - firstly the sorting routine. Secondly, testing whether the endpoints of the candidate edge belong to the same component or not requires Union-Find DS which incurs $(\log V)$ time per edge on an average. Now suppose there are t different weights of edges in the graph. We shall modify Kruskal in a way so that the overall runtime is $(t(E + V))$

Instead of sorting the edges, we run the algorithm in t phases - one phase considers edges of a particular weight only and we run them in increasing order of weights.

for implementing the algorithm in phases - each phase for a specific weight

Let the edge weights be $w_0 < w_1 < \cdots w_t$. We shall define an auxiliary graph G_p for phase $p = 0, 1, 2, \cdots t$. Let C_p be the set of connected components of the set of edges having weight less than w_p. Then, the vertex set V_p of G_p is just C_p. One may think of C_p as *contracted* to a single vertex in V_p.

The edge set E_p is the set of edges $(u, v) \in E$ with weight w_p which connect two vertices

in V_p i.e. u and v lie in different components in C_p.

G_0 is the graph with all vertices in G and edges of weight w_0. Notice that C_{p+1} (and hence V_{p+1}) is the set of connected components of G_p.

Algorithm Our algorithm finds a maximal forest of G_p for each p, and output the union of each of these forests.

To implement this, maintain a set of edges F which is initially empty. At any phase $p = 0, 1, 2 \cdots t$, perform the following operations

a) Find the connected components of G_p using BFS/DFS and add to F an arbitrary spanning tree of each component

b) Construct G_{p+1} as follows : V_{p+1} is the set of connected components of G_p. Construct an adjacency list representation of E_{p+1} by adding each edge $(u, v) \in E$ of weight w_{p+1} to the adjacency lists of the connected components of u and v whenever (u, v) belong to different vertices of V_{p+1}.

Explanation. *Correctness:* The above algorithm is essentially a simulation of Kruskal's algorithm while avoiding cycle detection using Union-Find. Kruskal's algorithm considers the edges in non-decreasing order of edge weights, which is the same as the order of our phases. Our algorithm works because of the following crucial property : after any phase p, the connected components of G_{p+1} correspond to the connected components in Kruskal's algorithm. Hence, the total cost of edges in phase p is the same for our algorithm and for Kruskal's, since all edges considered in the phase have the same weight.

We can prove the above property by induction. It is clearly true before the start of the first phase. For the induction step, fix a phase p. Before phase p begins, the connected components in Kruskal's algorithm are V_p. Then, Kruskal's algorithm picks a maximal set edges of E_p that do not form a cycle (i.e. a maximal forest) in V_p. Hence, in each connected component of G_p, Kruskal's algorithm picks a spanning tree, which is also what our algorithm picks. Therefore, Kruskal's and our algorithm have the same connected components after phase p.

Running time: Each phase takes $O(V + E)$ time to find connected components and to contract the connected components of G_p. Since there are t phases, the total time is $(t(V+E))$.

Tutorial 10

ADA

2023

Q1: Consider the following problem. You are given a flow network with unit capacity edges: It consists of a directed graph $G = (V, E)$, a source $s \in V$, and a sink $t \in V$; and $c_e = 1$ for every $e \in E$. You are also given a parameter k. The goal is to delete k edges so as to reduce the maximum $s-t$ flow in G by as much as possible. In other words, you should find a set of edges $F \subseteq E$ so that $|F| = k$ and the maximum $s-t$ flow in $G' = (V, E - F)$ is as small as possible subject to this.

Give a polynomial-time algorithm to solve this problem.

Answer: We assume that k is at most the max-flow otherwise flow can never be reduced by k. Observe that by removing any k edges in a graph, we reduce the capacity of any cut by at most k (since $c_e = 1; \forall e$), and so, the min-cut will reduce by at most k.

Therefore, by max-flow-min-cut theorem, the max-flow will reduce by at most k. In fact, we can show that max-flow can be reduced by k. To achieve this, we take a min-cut X (which obviously has at least k many cut edges) and remove k edges going out of it. The capacity of this cut will now become $f - k$, where f is the value of the max-flow. Therefore, the min-cut becomes $f - k$, and so, the max-flow becomes $f - k$. □

Q2: Give a polynomial time algorithm for the following minimization analogue of the Maximum Flow Problem. You are given a directed graph $G = (V, E)$, with a source $s \in V$ and sink $t \in V$, and numbers (capacities) $l(v, w)$ for each edge $(v, w) \in E$. We define a flow f, and the value of a flow, as usual, requiring that all nodes except s and t satisfy flow conservation. However, the given numbers are lower bounds on edge flow—that is, they require that $f(v, w) \geq l(v, w)$ for every edge $(v, w) \in E$, and there is no upper bound on flow values on edges.

Give a polynomial time algorithm that finds a feasible flow of minimum possible value.

Answer: First send a flow from s to t such that all lower bounds are satisfied. We can easily do this as follows: for every edge $e = (u, v)$ find a path from s to t which goes through (u, v), and send l_e amount of flow on this path. Let f denote this flow. Now that we have a feasible flow, we will try to reduce the flow as much as possible, but we do not want to reduce the flow on an edge below l_e. Therefore, we reverse every edge, and make it's capacity equal to $f_e - l_e$. Now we send a max flow from t to s in this "reversed" graph. Let $\bar{f}_e$ is the flow of edge e in $t - s$ max flow. The min flow from s to t is $f_e - \bar{f}_e$ for every edge e.

Here is a justification for the correctness. Consider the flow is not minimized. Then there exists a path from s to t such that the current flow on this path can be further minimized.

Let f be the initial feasible flow from s to t before reversing the flow. Now consider the reverse flow along this path. By the property of max flow, there must be an edge e on this path, which is saturated during the max flow. Thus reverse flow on this edge is $f - l_e$. So, in the final min flow, the flow of the e is $(f - (f - l_e)) = l_e$, i.e. flow is equal to the minimum load on e. Thus, the current flow can not be reduced further on this path.

□

Q3: Suppose you are given a flow network G with integer edge capacities and an integer maximum flow f^* in G. Describe algorithms for the following operations:

1. Increment(e): Increase the capacity of edge e by 1 and update the maximum flow.
2. Decrement(e): Decrease the capacity of edge e by 1 and update the maximum flow.

Both algorithms should modify f^* so that it is still a maximum flow, more quickly than recomputing a maximum flow from scratch.

Answer: (a) If there exists a minimum cut on which (u, v) doesn't lie then the maximum flow can't be increased, so there will exist no augmenting path in the residual network. Otherwise, it does cross a minimum cut, and we can possibly increase the flow by 1. Perform one iteration of Ford-Fulkerson. Compose the residual graph on the original flow. Add a positive 1 capacity on the edge that has been increased. Using BFS, search for an augmenting path. If there exists an augmenting path, it will be found and increased on this iteration. Since the edge capacities are integers, the flow values are all integral. Since flow strictly increases, and by an integral amount each time, a single iteration of Ford-Fulkerson will increase the flow by 1, which we know to be maximal. To find an augmenting path we use a BFS, which runs in $O(|V| + |E'|) = O(|V| + |E|)$.

(b) If the edge's (say $e = (u, v)$) flow was already at least 1 below capacity then nothing changes. Otherwise, observe that $f_e = c_e$. Now, flow through the edge is decreased by 1. To decrease the flow find a path from s to t which contains (u, v) using BFS in $O(V + E)$. To do this, just run two BFS, one starting from s to u going through edges that have flow greater than 0, the other from v to t equally from edges that have flow greater than 0. (It is guaranteed you will reach your goal in both cases since there is a unit of flow traversing this edge). Select a path from s to u and remove 1 unit of flow from each edge and do the same with a path from v to t. This decreases total flow by 1.

Observe that decreasing flow through (u, v) does not mean that the max flow necessarily decreased. Thus we need to recover the flow. After decreasing the total flow, now we have a valid flow network. Now we just need to run one iteration of the augmenting path with the reduced capacity of edge e (Just like running one iteration of Ford-Fulkerson algorithm). Running one iteration has the complexity of a single DFS/BFS which is $O(|V| + |E|)$. By the argument given in part a, everything is integer-valued and flow strictly increases, so we will either find no augmenting path, or will increase the flow by 1 and then terminate.

Winter 2023 Analysis and Design of Algorithms: MidSem Solutions *Max: 60 Marks*
Feb 27, 2023 *120 minutes*

Part A

For each of the following, choose the correct answer.

1. **(7 Marks)** Given three algorithms:

   - **Algorithm A:** Partitions a problem of size n into 8 subproblems of size $n/3$ and combines them in time $O(n^2)$.
   - **Algorithm B:** Partitions a problem of size n into 4 subproblems of size $n/2$ and combines them in time $O(n^2 \log n)$.
   - **Algorithm C:** Partitions a problem of size n into 4 problems of size $n/4$ and combines them in time $O(n)$.

   The asymptotic running times of the algorithms in increasing order are:

   (a) A, B, C
   (b) A, C, B
   (c) B, A, C
   (d) B, C, A
   (e) C, A, B (**correct**)
   (f) C, B, A
   (g) All have the same asymptotic running time.

   Solution: For the Algorithm A, the recurrence is $T(n) = 8T(n/2) + O(n^2)$. It means that the algorithm A takes $O(n^3)$-time. For the algorithm B, the recurrence is $T(n) = 4T(n/2) + O(n^2 \log^2 n)$. It means that the algorithm B takes $O(n^2 \log^2 n)$. On the other hand, for algorithm C, the recurrence is $T(n) = 4T(n/4) + O(n)$. It means that the algorithm takes $O(n \log n)$-time. So, the running times at increasing order is C, A, B.

2. **(7 Marks)** Give an asymptotically tightest upper bound in Big-Oh notation for this recurrence relation $T(n) = 4T(\sqrt{n}) + (\log n)^2$. Assume $T(n)$ is constant for $n \leq 2$.

   (a) $O(\log(\log n)^2 \log n)$
   (b) $O((\log n)^2 \log\log n)$ (**Correct**)
   (c) $O((\log n)^2 \log n)$
   (d) $O(\log n \log\log n)$

   Solution: We use variable change method for this. Let $n = 2^k$. Hence, $\log n = k$. Then, this recurrence can be represented as $S(k) = 4S(k/2) + O(k^2)$. Solving this recurrence gives $O(k^2 \log k)$. Hence, $T(n)$ is $O((\log n)^2 \log\log n)$.

3. **(5 Marks)** Suppose that there are two strings $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$. Let $L[i, j]$ denotes the length of a longest common subsequence between $\langle x_1, \ldots, x_i \rangle$ and $\langle y_1, \ldots, y_j \rangle$. If $x_i = y_j$, then recurrence relation for the dynamic programmin to solve $L[i, j]$ is

   (a) $L[i, j] = \max\{L[i, j-1], L[i-1, j]\}$
   (b) $L[i, j] = 1 + \max\{L[i, j-1], L[i-1, j]\}$
   (c) $L[i, j] = 1 + L[i-1, j-1]$. (**Correct**)
   (d) None of the above.

4. **(6 Marks)** Let $LC(n)$ be the sequence of *lucas numbers* defined by $LC(0) = 2, LC(1) = 1$ and for all $n \geq 2$, $LC(n) = LC(n-1) + LC(n-2)$. What is the worst case running time of the fastest algorithm to compute $LC(n)$? Just right down the tightest possible asymptotic running time in Big-Oh notation.

   (a) $O(1.618^n)$.

   (b) $O(n \log n)$.

   (c) $O(n)$.

   (d) $O(n^2)$.

   Solution: Computing this can be built iteratively. Initialize $Lucas[0] = 2$ and $Lucas[1] = 1$. Then, for every $i = 2, \ldots, n$ in this order, $Lucas[i] = Lucas[i-1] + Lucas[i-2]$. Hence, this can be computed in $O(n)$-time.

Part B

1. **(10 Marks)** Given n objects, how many different sets of size k can be chosen? Write down the pseudocode of an algorithm. Give an explanation of the running time of the algorithm. ANY ALGO AND COMPATIBLE COMPLEXITY IS FINE. DP IS DESIRABLE; PSEUDOCODE/STEPS

   - **Number of different sets of size k:** (2 Marks) $\binom{n}{k}$.
   - **Pseudocode of the algorithm:** Here is a basic result from discrete mathematics (not for grading).

$$\text{For all } n > k \text{ and } k \geq 1, \text{ it holds that } \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

   If $n = k$, then $\binom{n}{k} = 1$ and if $k = 0$, then $\binom{n}{k} = 1$.

   The pseudocde works as follows. We describe both the recursive algorithm and iterative algorithm. Both carry equal marks. One student would write one of them. Then, give marks accordingly.

   Recursive algorithm: (5 Marks)

Algorithm 1: Computing the number of subsets of size k

```
Data: Function Count(n, k)
if n < k then
    return 'undefined';
if k = n or k = 0 then
    return 1;
Return Count(n − 1, k − 1) + Count(n − 1, k);
```

   Iterative Algorithm: (5 Marks)

Algorithm 2: Computing the number of subsets of size k

```
Data: Function Count(n, k)
if n < k then
    return 'undefined';
if n = k or k = 0 then
    Return 1;
for i = 1, ..., n do
    Count[i, i] = 1; and Count[i, 0] = 1;
for i = 2, ..., n do
    for j = 1, ..., min(i − 1, k) do
        Count[i, j] = Count[i − 1, j − 1] + Count[i − 1, j];
Return Count[n, k];
```

- Running time of the algorithm: (3 Marks)
  Running time of the recursive algorithm has the recurrence

$$T(n,k) = T(n-1,k-1) + T(n-1,k)$$

  The iterative algorithm builds up table in the bottom-up fashion. Hence, the running time of the algorithm is $O(nk)$.
  (1 Mark for mentioning the correct running time and 2 Marks for justification)

2. (10 **Marks**) In this problem we consider a monotonously decreasing function $f : N \to Z$ (that is, a function defined on the natural numbers taking integer values, such that $f(i) > f(i+1)$). Assuming we can evaluate f at any i in constant time, we want to find $n = \min\{i \in N | f(i) \leq 0\}$ (that is, we want to find the value where f becomes negative). Write the pseudo-code of an $O(\log n)$-time algorithm and give a justification why your algorithm runs in $O(\log n)$-time. **The solution lies in [a,b] s.t. a,b \in N; there is a guarantee that the sol will be converged at**

   - **Pseudocode of the Algorithm:** (7 Marks).

Algorithm 3: Finding out n

```
Data: Input is f : N → Z
Initialize k = 0 and r_s = 0;
while do
    r_s ← r_s + 2^k and r_m ← r_s + 2^{k+1};
    if r_m = r_s + 1 and f(r_s) > 0 but f(r_m) ≤ 0 then
        return r_m;
    if f(r_s), f(r_m) > 0 then
        k ← k + 1;
    else
        if f(r_m) = 0 then
            Return r_m;
        else
            (it must be that f(r_s) > 0 but f(r_m) < 0)
            j ← Binary-Search(upper = r_s, lower = r_m, f);
            (this above function returns an index i that is at least r_s and at most r_m such that f(i) = 0 if
             exists. It returns i with f(i) < 0 if f(i − 1) > 0)
            Return j;
```

The binary search subroutine described above does a binary search with lower index r_s and upper index r_m and at every stage, checks if $f(i) \leq 0$ but $f(i) > 0$.

- **Explanation of the running time of your algorithm:** (3 Marks) Observe that the binary search subroutine is used with lower index r_s and upper index r_m. Also, $r_m - r_s = 2^k$ and $r_m = 2r_s$. As $2^k = r_m - r_s$, it only requires to invoke $f(i)$ for $k+1$ distinct values. Then, the binary search runs in $O(k)$-time in for this range of values. Also, $n \leq 2^{k+1}$.
  Hence, the running time of the algorithm is $O(k)$, i.e. $O(\log n)$.

  (deduct 1 Mark if the explanation is partially correct. If the justification is absent, then deduct 2 Marks)

3. **(20 Marks)** Suppose that you are running a large computing job in which you need to simulate a physical system for as many discrete steps as you can. The lab you are working as two large supercomputers A and B. But your job can only run one of the machines at any given minute. Over each of the next n minutes, you have a 'profile' of how much processing power is available on each machine. In minute i, you would be able to run $a_i > 0$ steps if your job uses machine A and $b_i > 0$ steps of simulation if your job uses machine B. ou can move your job from one machine to the other. But moving job from one machine from other costs you a minute and no processing can be done in that minute. So, given a sequence of n minutes, a *plan* is specified by a choice A, B or *move* for each minute with the property that two distinct machines cannot appear in two consecutive minutes. It means that if you use machine A in minute i and you want to switch to machine B, then choice for minute $i+1$ must be *move*. The *value* of a plan is the total number of steps that you manage to execute over the n minutes: so, it's the sum of a_i over all the minutes in which the job is on A, plus the sum of b_i in which your job is in B. Design a dynamic programming based algorithm to compute the the value of an *optimal plan* (i.e. a plan with maximum value). **OVER n MINUTES**

- **Define the sub-problems and state the number of sub-problems** (4 Marks)
  For minutes $\{1, \ldots, k\}$, we define two subproblems.
  $StepA(k)$ is the maximum value of a plan if machine A is used at the k-th minute.
  $StepsB(k)$ is the maximum value of a plan if machine B is used at the k-th minute.
  Hence, there are 2 subproblems. If somebody also adds $ST(k) = \max\{StepsA(k), StepsB(k)\}$ then also no deduction of marks.
  Rubric: If somebody writes the subproblem definitions correctly, but his/her number of subproblems mentioned is inconsistent, then deduct one mark.
- **The recurrence relation including base case(s):** (6 Marks)
  Base cases are: $StepsA(1) = a_1$ and $StepsA(2) = a_1 + a_2$; $StepsB(1) = b_1$ and $StepsB(2) = b_1 + b_2$. For $k \geq 3$, the recurrence is

$$StepsA(k) = a_k + \max\left\{StepsA(k-1), StepsB(k-2)\right\}$$

$$StepsB(k) = b_k + \max\left\{StepsB(k-1) + StepsA(k-2)\right\}$$

  (**Rubric:** 2 Marks for base cases and 4 Marks for correct recurrences. Deduct 1 mark if partially incorrect base case and deduct 2 additional marks if partially correct recurrence).
- **The subproblem that solves the actual problem:** (2 Marks)

$$\max\{StepsA(n), StepsB(n)\}$$

  .
- **A brief description dynamic programming algorithm**: (5 Marks) We use two different arrays $A[1, \ldots, n]$ and $B[1, \ldots, n]$ as follows.
  - Initialize $A[1] = a_1$, $A[2] = a_1 + a_2$, $B[1] = b_1$ and $B[2] = b_1 + b_2$.
  - For $k = 3, \ldots, n$; assign $A[k] = a_k + \max\{B[k-2], A[k-1]\}$ and assign $B[k] = b_k + \max\{B[k-1] + A[k-2]\}$.
  - Finally, output $\max\{A[n], B[n]\}$.

  (**Rubric:** deduct 2 marks for partially correct algorithm description or partially correct pseudocode)
- **Explanation of the running time**: (3 Marks) Initialization of $A[1], A[2], B[1], B[2]$ values require $O(1)$-time. Computing $A[k]$ requires $O(1)$ many comparison operations and other arithmetic operations. Hence, the second step requires $O(n)$-time. Finally, the output requires $O(1)$-time. Hence, the running time of the algorithm is $O(n)$.
  (**Rubric:** deduct 1 mark for incorrect justification while the running time is correct and consistent with the algorithm)

Winter 2023
May 4, 2023

ADA Endsem Rubrics

Max: 60 Marks
120 minutes

Roll No: _____ **Section:** __

Name: _____

Please write solutions independent of each other. This is a cheat based examination. You can carry **two A4-sized reference sheets** *where you can have some formulas etc written. Those two A4-sized reference sheets must contain your roll number and section and they have to be submitted along with the answer booklet. Moreover, those cheat-sheets cannot be shared with others. Furthermore, your solution must fit in the space provided. Extra sheet will be provided* only *for rough work . So try to be precise and brief. Meaningless blabber fetches negative credit. Also, you can use anything as a subroutine that was taught in the lectures or tutorials.*

Part A

Question	1	2	3	4	5	6	7	8	9	10	Total
Marks											

Part B

Question	1	2	3	4	Total
Marks					

Total Marks:

Part A

This part has 10 questions. Answer any 8 questions.

1. **(3 Marks)** Let G be a directed graph having $s, t \in V(G)$ and positive edge weights. Consider a longest simple path P (i.e. a path P with maximum total edge weight) from s to t in G. Let $u \in P$ be an internal vertex in this path P. Your friend claims that the subpath of P from s to u is a longest simple path from s to u in G. Is this claim **TRUE** or **FALSE**? You do not need to justify your answer.

   Answer : FALSE.

   Consider the graph with the following edges $(s, 1)$, $(s, 2)$, $(1, 4)$, $(2, 3)$, $(3, 4)$, $(4, 2)$, $(4, t)$. The longest path form s to t is $s - 2 - 3 - 4 - t$. Longest path form s to 2 is $s - 1 - 4 - 2$.

   Rubric : +3 for correct / **0** for incorrect.

2. **(3 Marks)** Which of the following statements (**it could be more than one**) is/are **TRUE** for an undirected graph?
   Statement P: Number of odd degree vertices is even.
   Statement Q: Sum of degrees of all vertices is even

   (a) Only statement P.
   (b) Only statement Q.
   (c) Both statements P and Q.
   (d) Neither statement P nor statement Q.

   Answer : (c)

   Q is true: Since the graph is undirected, every edge increases the sum of degrees by 2. P is true: If we consider sum of degrees and subtract all even degrees, we get an even number (because Q is true). So total number of odd degree vertices must be even.

   Rubric : +1 for (a) / **+1** for (b) / **+3** for (c) / **0** for (d)

3. **(3 Marks)** Consider an edge weighted undirected connected graph G such that all edge weights are positive and distinct. Let $e_{\min}$ and $e_{\max}$ be the edges with smallest weight and largest weight in G respectively. Then which of the following statements is **FALSE**?

   (a) Every minimum spanning tree of G must contain $e_{\min}$.
   (b) If $e_{\max}$ is in a minimum spanning tree, then its removal must disconnect G.
   (c) No minimum spanning tree contains $e_{\max}$.
   (d) G has a unique minimum spanning tree.

   Answer : (c)

   As edge weights are unique, there will be only one edge $e_{\min}$ and that will be added to MST, therefore option (A) is always true. As spanning tree has minimum number of edges, removal of any edge will disconnect the graph. Therefore, option (B) is also true. As all edge weights are distinct, G will have a unique minimum spanning tree. So, option (D) is correct. Option C is false as $e_{\max}$ can be part of MST if other edges with lesser weights are creating cycle and number of edges before adding $e_{\max}$ is less than $(n - 1)$.

   Rubric : +3 for (c) / **0** for others

4. **(3 Marks)** What is the time complexity of the following recurrence relation? $T(n) = 8T(\sqrt{n}) + (\log n)^2$? The base case of this recurrence relation is $T(n) = 1$ for $n \leq 2$. Give the tightest possible $f(n)$ in Big-Oh notation. Please write down the answer only. An explanation is not needed.

   Answer : $T(n) = \theta(\log_2 n)^3$

   Given $T(n) = 8T(\sqrt{n}) + (\log n)^2$. Let's take $n = 2^m$ Then , $T(2^m) = 8T(2^{m/2}) + m^2$. Let's take $T(2^m)$ as $S(m)$, $S(m) = 8S(m/2) + m^2$. Apply Master's Theorem, $a = 8, b = 2, d = 2$. By case: $d < \log_b a$, $S(m) = O(m^3)$. Put m as $\log_2 n$. Thus, $T(n) = O(\log_2 n)^3$.

Rubric : Any process of proving is acceptable. Steps should be clear and correct. **+1** for any correctly proven loose bound give. **0** for incorrect steps and loose bound give. **+2** for minor mistakes and loose bound give. **+3** clear steps and tight bound. **0** very unclear steps or incorrect bound give.

5. (3 **Marks**) There are five students in a class namely X, Y, Z, W, and V. Student X claims that a triangle is a bipartite graph. Student Y claims that a pentagon is a bipartite graph. Student Z claims that a square is a bipartite graph. W claims heptagon is a bipartite graph. Student V claims that a decagon is a bipartite graph. Who among them is/are correct? (for instance, if you think X and Z are saying correct but not the rest, then you should write "only X and Z are saying correct statement")

   Answer : Only V and Z are saying the correct statement.

   The argument follows from the fact that even length cycles are bipartite graph, because they can be colored by two colors. But for odd length cycles, we require 3 colors, thus not bipartite graph.

   Rubric : +1 for one correct and one incorrect answer / **+2** for two correct and one incorrect answer /**+2** for one correct answer and no incorrect answer / **+3** for all correct answers and no incorrect answer / **0** for more than one incorrect answer.

6. (3 **Marks**) Let G be an undirected graph. Consider a depth-first traversal of G starting from s and let T be the resulting depth-first search tree rooted at s. Let u be a vertex in G and let v be the first new (unvisited) vertex visited after visiting u in the traversal. Which of the following statements is always **TRUE**?

   (a) (u, v) must be an edge in G, and u is a descendant of v in T.
   (b) (u, v) must be an edge in G, and v is a descendant of u in T.
   (c) If (u, v) is not an edge in G then u is a leaf in T.
   (d) If (u, v) is not an edge in G then u and v must have the same parent in T.

   Answer : (c)

   In DFS, if v is visited after u, then either (u, v) is an edge or u is a leaf node. In DFS after visiting a node, we first recur for all unvisited children. If there are no unvisited children (u is leaf), then control goes back to parent and parent then visits next unvisited children.

   Rubric : +3 for correct / **0** for incorrect.

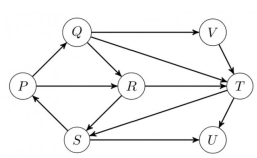
7. (3 **Marks**) Every directed acyclic graph has exactly one topological ordering. Is this **TRUE** or **FALSE**? Explain.

   Answer : FALSE.

   Some priority constraints may be unspecified, and multiple orderings may be possible for a given DAG. For example a graph $G = (V, E) = (\{a, b, c\}, \{(a, b), (a, c)\})$ has valid topological orderings $[a, b, c]$ or $[a, c, b]$.

   Rubric : +3 for correct / **0** for incorrect.

8. (3 **Marks**) Which of the following is the correct decomposition of the directed graph given below into its strongly connected components?



   (a) {P, Q, R, S, T, U, V}
   (b) {P, Q, R, S, T, V}; {U}
   (c) {P, Q, S, T, V}; {R}; {U}
   (d) {P, Q, R, S}; {T}; {U}; {V}

   Answer : (b)

A graph is said to be strongly connected if every vertex is reachable from every other vertex. The strongly connected component is always maximal that is if x is strongly connected component there should not exist another strongly connected component which contains x. If we take R as a strongly connected component but which is part of $PQRS$ and $PQRS$ is part of $PQRSVT$.

Rubric : +3 for correct / **0** for incorrect.

9. (3 **Marks**) Consider a simple undirected weighted graph G all of whose edge weights are distinct. Which of the following statements about the minimum spanning trees of G is/are **TRUE**?

   (a) The edge with the second smallest weight is always part of any minimum spanning tree of G.

   (b) One or both of the edges with the third smallest and the fourth smallest weights are part of any minimum spanning tree of G.

   (c) Suppose $S \subseteq V$ be such that $S \neq \varnothing$ and $S \neq V$. Consider the edge with the minimum weight such that one of its vertices is in S and the other in $V \setminus S$. Such an edge will always be part of any minimum spanning tree of G.

   (d) G can have multiple minimum spanning trees.

   Answer : (a), (b) and (c)

   Rubric : 0 for any incorrect answer / **+1** for one correct answer / **+2** for two correct answer / **+2** for all correct answers

10. (3 **Marks**) Show: $O(\log_2 N) = O(\log_7 N)$.

    Answer : This is an exercise of changing base. We know,

$$\log_a b = \frac{\log_c b}{\log_c a}$$

So we can write,

$$\log_7 N = \frac{\log_2 N}{\log_2 7}$$

Since $\log_2 7$ is a constant, we have $O(\log_2 N) = O(\log_7 N)$

Rubric : +3 for correct proof / **+1** for partial steps or directly writing $\log_7 N$ is constant factor multiplicative of $\log_2 N$ / **0** for incorrect proof

Part B

This part has 4 questions. Answer any 3 questions.

1. **(12 Marks)** Given an $m \times n$ binary matrix whose 0 represents water and 1 represents land. The connected 1s form an island. Formally, a cell containing 1 with indices (i, j) (i indicates row and j indicates column) is considered connected to either of the following cells if that cell contains 1: $(i-1, j)$, $(i-1, j-1)$, $(i, j-1)$, $(i, j+1)$, $(i+1, j)$, $(i-1, j+1)$, $(i+1, j-1)$, and $(i+1, j+1)$. The below figure depicts a problem instance. Here the light-colored cells have 1s, and dark-colored cells have 0s. There are a total of 5 islands as indicated by digits 1 - 5. Design an $O(mn)$-time algorithm to count the number of islands.

1		2				3	3	3	3
		2		2		3			
2	2	2	2			3			
2			2		3				
2	2	2	2				5	5	5
	2		2			5	5	5	5
					5	5	5		
			4			5	5	5	
4		4		4			5		
4	4	4	4				5	5	5

<u>Graph construction:</u>

(**+2** for identifying the problem is to find connected component) : The solution is inspired by finding the total number of connected components in a graph problem. The idea is to start Breadth–first search (BFS) from each unprocessed node and increment the island count. Each BFS traversal will mark all cells which make one island as processed. So, the problem reduces to finding the total number of BFS calls.

(**+3** for identifying the proper vertices and edges corresponding to matrix) : Before BFS we have to convert the given matrix in to graph. We consider each cell marked 1 as vertex and and put an edge between each neighbouring cells if the neighbour is marked 1. A cell has at-most 8 neighbouring cells and at-least 3 neighbouring cells. Thus the complexity of the graph is bounded by $m \times n$.

(**+4** for showing the construction, specifically identifying the 8 adjacent cells for edges) : Let I be the given matrix, So the construction for graph G is as follows:

(a) $G(V) \leftarrow \{\phi\}$; $G(E) \leftarrow \{\phi\}$

(b) For each (i, j) pair
If $I_{i,j} = 1$ then $V \leftarrow V \cup \{v_{i,j}\}$

(c) For each $v_{i,j} \in V$
If $v_{i-1,j-1} \in V$ then $E \leftarrow E \cup \{v_{i,j}, v_{i-1,j-1}\}$
Else if $v_{i-1,j} \in V$ then $E \leftarrow E \cup \{v_{i,j}, v_{i-1,j}\}$
Else if $v_{i-1,j+1} \in V$ then $E \leftarrow E \cup \{v_{i,j}, v_{i-1,j+1}\}$
Else if $v_{i,j-1} \in V$ then $E \leftarrow E \cup \{v_{i,j}, v_{i,j-1}\}$
Else if $v_{i,j+1} \in V$ then $E \leftarrow E \cup \{v_{i,j}, v_{i,j+1}\}$
Else if $v_{i+1,j-1} \in V$ then $E \leftarrow E \cup \{v_{i,j}, v_{i+1,j-1}\}$
Else if $v_{i+1,j} \in V$ then $E \leftarrow E \cup \{v_{i,j}, v_{i+1,j}\}$
Else if $v_{i+1,j+1} \in V$ then $E \leftarrow E \cup \{v_{i,j}, v_{i+1,j+1}\}$

Now we are ready for the algorithm.

Steps of the algorithm:

(**+3** for correctly computing number of connected components, with the help of a counter)

(a) Initialize an empty queue Q. Also set a counter C to be 0.

(b) While doing BFS on G using Q

- If Q is empty
  $C \leftarrow C + 1$

(c) Output C

2. (12 **Marks**) Given an $m \times n$ matrix of characters, find the length of the longest path in the matrix starting from a given character. All characters in the longest path should be increasing and consecutive to each other in alphabetical order. We are allowed to search the string in all eight possible directions, i.e., North, West, South, East, North-East, North-West, South-East, South-West. For example, consider the following matrix of characters:

```
[ D, E, H, X, B ]
[ A, O, G, P, E ]
[ D, D, C, F, D ]
[ E, B, E, A, S ]
[ C, D, Y, E, N ]
```

The length of the longest path with consecutive characters starting from character C is 6. The longest path is: C(2, 2) → D(2, 1) → E(3, 2) → F(2, 3) → G(1, 2) → H(0, 2)

[**Hint:** DP on DAG]

As the hint suggested, we can view the problem as finding the longest the longest path form a given node in the DAG.

The creation of DAG is straight forward. For each cell (i, j) of the matrix we consider a vertex $v_{i,j}$ in our graph like the last question. Now for each cell we check all of it's neighbouring cells. We put a directed edge from vertex $v_{i,j}$ to vertex $u_{k,l}$ if (k, l) is neighbour cell of (i, j) such that the character in (i, j) is smaller than the character in (k, l) in alphabetical order. Observe that the created graph say G is a DAG by definition as any path in the graph maintains alpabetical ordering. Now the problem is to find the longest path for a given node in a DAG.

Note that, if a neighbour of a cell contains the same character as the cell, we will put directed edge between them to any of the direction. This may effect the optimal solution, but this case is acceptable for scoring.

(**+4** for subproblem)

- Subproblem definition:.

  For a given vertex v let $ch(v)$ denotes children of v. $LP(v)$ denotes the longest path starting form v in the graph $\forall v \in V(G)$.

  (**+3** for recurrence)

- Recurrence Relation:

$$LP(v) = \max\{LP(v), 1 + \max_{u \in ch(v)} \{LP(u)\}\}$$

  Base case : $LP(u) = 0; \forall u \in V : ch(u) = \{\phi\}$

(**+2** for this section)

- Subproblem that solves the actual problem:
  For a given cell (i, j); $LP(v_{i,j})$ gives the actual solution.

(**+3** for algorithm)

- Description of the algorithm:
  (a) Create a graph as discussed above.
  (b) For a given cell (i, j), return $LP(v_{i,j})$

  LP(v)

  (a) If $ch(v) = \{\phi\}$
      - return 0
  (b) $len \leftarrow 0$
  (c) For each $u \in ch(v)$ then
      - $len \leftarrow \max(len, 1 + LP(u))$
  (d) return len

  Note : The problem can be solved without using the concept id DAG. The DP can be defined over the matrix directly. However, the solution sketch remains the same and instead of children, one can use the concept of adjacent cells directly. Then the direction of movement needs to be clearly defined as there is no directed edge defined on the matrix directly.

3. **(12 Marks)** There are n teams that plan to attend a grand dinner in a restaurant. Team i has t_i members. The restaurant can provide m tables for the dinner, with $m \geq n$. Table i can has s_i chairs. We wish to seat all the teams such that no two team members are at the same table, so that we maximum students getting to meet members of other teams. Can we do so?
Model this as a network flow problem. Specify the vertices, edges and capacities. Then finally, explain how you can transform the maximum flow into an optimal solution to this problem.

- Flow network construction (including vertices, edges, capacities):
  (**+4** for proper construction of the graph. Any correct drawing of graph with little elaboration is also accepted) : Create a flow network with $n + m + 2$ vertices. Create one vertex for each team and one for each table. Create extra source and sink vertices. Create edges from the source to each team with a capacity of t_i. Create edges from each table vertex to the sink vertex with capacity s_i. Finally, add edges from each team to each table, with capacity 1, since each team can provide at most one person per table.

- How does the maximum-flow of your network can be transformed into an optimal solution to this problem?
  (**+3** for the point. **0** if not mentioned correctly) : Run the network flow algorithm. If the maximal flow equals the sum of the number of team members, the seating can be done. Otherwise, it can not be.

- Algorithm Description:
  (**+2** for this section)
  (a) Create the graph as explained above
  (b) Use the Ford Fulkerson's algorithm to get the max-flow from s to t for the above graph.
  (c) If the max flow is equal to the sum of the team members, output TRUE, else output FALSE.

  (**+3** for this section / **+1** if it is not mentioned that the algorithms runs in polynomial time, i.e. the maximum flow is polynomial w.r.t input graph)
- Explanation of the running time: The running time of the algorithm depends on the creation of the graph and the running time of Ford Fulkerson's algorithm. Noticed the induced graph between teams and table can be a complete bipartite graph. So the edge complexity of the graph can be $O(mn)$ in the worst case. The running time of Ford Fulkerson's algorithm is $O(|E| \times f)$ where $|E|$ is the number of edges in the graph and f is the maximum flow. Now here, since the maximum flow here can be maximum $O(mn)$, the total complexity of the graph is $O(m^2n^2)$.

4. **(12 Marks)** Define the *decision versions* of the following optimization problems. Each parts **(a), (b), (c)** have 4 marks.

(a) DOMINATING SET: Given an undirected graph $G = (V, E)$, find a set S with minimum number of vertices such that for every vertex $v \in V(G)$, either $v \in S$ or there is $u \in S$ such that $(u, v) \in E(G)$.

Answer : The decision version of the problem is follows. Given an undirected graph G and an integer k, the problem is to find if there exists a set S of vertices with $|S| \leq k$ such that for every vertex $v \in V(G)$, either $v \in S$ or there is $u \in S$ such that $(u, v) \in E(G)$.

Rubric : +4 for correct / **0** for incorrect.

(b) CLIQUE: Given an undirected graph $G = (V, E)$, find a set S with maximum number of vertices such that for every $u, v \in S$, $(u, v) \in E(G)$.

Answer : The decision version of the problem is follows. Given an undirected graph G and an integer k, the problem is to find if there exists a set S of vertices with $|S| \geq k$ such that for every $u, v \in S$; $(u, v) \in E(G)$.

Rubric : +4 for correct / **0** for incorrect.

(c) CLUSTER VERTEX DELETION SET: Given an undirected graph $G = (V, E)$, find a set S with minimum number of vertices such that every connected component of $G - S$ (i.e. after removing the vertices of S and the edges incident to S) is a clique (what it means is that if C is a connected component of $G - S$, then for every $u, v \in C$, $(u, v) \in E(G)$).

Answer : The decision version of the problem if follows. Given an undirected graph G and an integer k, the problem is to find a set S of vertices with $|S| \leq k$ such that every connected component of $G - S$ is a clique.

Rubric : +4 for correct / **0** for incorrect.

Quiz-1 (ADA-2023)

February 2, 2023

1. Let $f(n) = 25^n n^3$ and $g(n) = 36^n n$. Then which of the following statement(s) is/are true?

(A) $g(n) = O(f(n))$.

(B) $f(n) = O(g(n))$. (correct)

(C) Both the above.

(D) None of the above.

2. Let $f(n) = \log_{20} n$ and $g(n) = \log_5 n$. Then which of the following statement(s) is/are true?

(A) $g(n) = O(f(n))$.

(B) $f(n) = O(g(n))$.

(C) Both the above. (correct)

(D) None of the above.

3. Let $f(n) = 6^{2\log_6 n}$ and $g(n) = n^2$. Then which of the following statement(s) is/are true?

(A) $g(n) = O(f(n))$.

(B) $f(n) = O(g(n))$.

(C) Both the above. (correct)

(D) None of the above.

4. Let $f(n) = 2^n n^9$ and $g(n) = 2^n n^7$. Then which of the following statement(s) is/are true?

(A) $g(n) = O(f(n))$. (correct)

(B) $f(n) = O(g(n))$.

(C) Both the above.

(D) None of the above.

5. Suppose that an algorithm $\mathcal{A}$ partitions a problem of size n into 7 subproblems each of size $n/3$ and then combines the solutions in $6n^2$-time. When $n \leq 6$, then it takes only 4 primitive operations. Then what is the recurrence relation of algorithm $\mathcal{A}$?

(A) $T(n) = 6T(n/3) + 6n^2$ for all $n \geq 7$ and $T(n) = 2$ for all $n \leq 6$.

(B) $T(n) = 6T(n/3) + 6n^2$ for all $n \geq 2$ and $T(n) = 4$ for all $n \leq 6$..

(C) $T(n) = 7T(n/3) + 6n^2$ for all $n \geq 2$ and $T(n) = 2$ for all $n \leq 6$..

(D) $T(n) = 7T(n/3) + 6n^2$ for all $n \geq 2$ and $T(n) = 4$ for all $n \leq 6$.. (correct)

6. Suppose that an algorithm $\mathcal{A}$ partitions a problem of size n into 16 subproblems each of size $n/4$ and then combines the solutions in $64n^2$-time. Then what is the tightest asymptotic running time of algorithm $\mathcal{A}$ in Big-Oh notation?

(A) $O(n \log n)$. (B) $O(n^2)$. (C) $O(n^2 \log n)$. (D) $O(n^3)$.

Option C

7. Suppose that an algorithm $\mathcal{A}$ partitions a problem of size n into 4 subproblems each of size $n/4$ and then combines the solutions in $2n \log n$ time. Then what is the tightest asymptotic running time of algorithm $\mathcal{A}$ in Big-Oh notation?

(A) $O(n \log n)$. (B) $O(n^2)$. (C) $O(n(\log n)^2)$. (D) $O(n^2 \log n)$.

Option C.

8. Suppose that an algorithm $\mathcal{A}$ partitions a problem of size n into 6 subproblems each of size $n/2$ and then combines the solutions in $6n^3$ time. Then what is the tightest asymptotic running time of algorithm $\mathcal{A}$ in Big-Oh notation?

(A) $O(n^2 \log n)$. (B) $O(n^2)$. (C) $O(n^3 \log n)$. (D) $O(n^3)$.

Option D.

9. Consider the following recurrence.

$$T(n) = T(n/4) + T(2n/5) + 1 \text{ for all } n \geq 6$$

$$T(n) \leq 10 \text{ for all } n \leq 5$$

Then, what is $T(n)$? Give the tightest possible function $f(n)$ using Big-Oh notation.

(A) $O(n)$.

(B) $O(\log n)$.

(C) $O(\sqrt{n})$.

(D) None of the above.

Option A.

10. What is the solution to the recurrence relation $T(n) = T(n-1) + n^3$ and $T(1) = 1$ in Big-Oh notation? Give the tightest possible running time.

(A) $O(n^4 \log n)$. (B) $O(n^4)$. (C) $O(n^3 \log n)$. (D) $O(n^3)$.

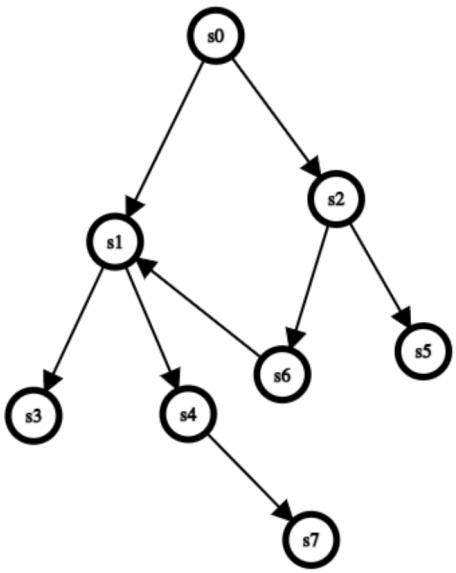
Option B

Analysis and Design of AlgorithmsRubrics
Quiz-2
Set-1

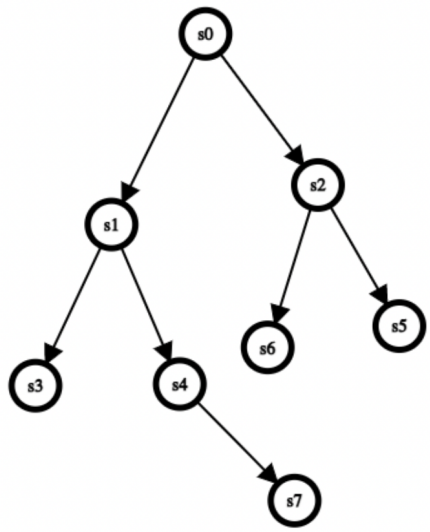
Date: 23.02.2023

Question 1)

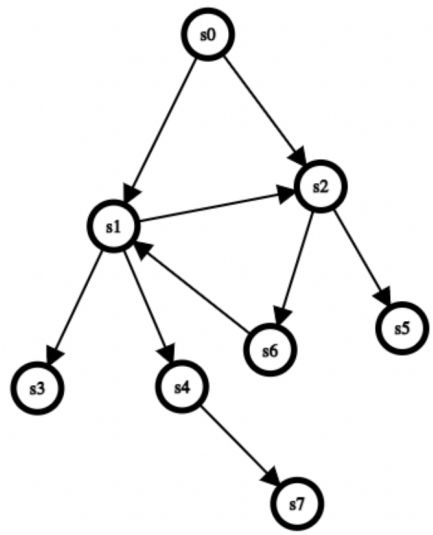
Recurrence 1



Recurrence 2



Recurrence 3



Given 3 Recurrences, to evaluate the s0 for each of the cases **optimally** which of the following is correct

a) 1->Recursion, 2->Dynamic Programming, 3->Recursion
b) 1->Can't be Evaluated using Rec or DP, 2->DP, 3->DP
c) 1->DP, 2->Recursion, 3->Recursion
d) 1->DP, 2->Recursion, 3->Can't be Evaluated using Rec or DP

Question 2)

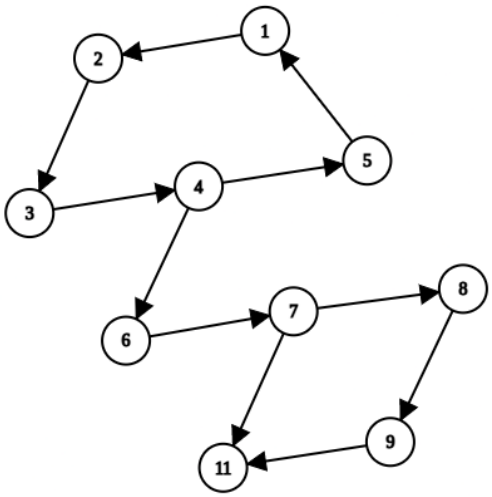
Consider a country whose coins are minted with denominations of {d1, . . . , dk } units. We want to count how many distinct ways C(n) there are to make change of n units. For example, in a country whose denominations are {1, 6, 10}, C(5) = 1, C(6) to C(9) = 2, C(10) = 3, and C(12) = 4. Now, How many ways are there to make change of 20 units from {1, 6, 10}?

A. 20

B. 1
C. 7
D. 11
E. 10

Question 3)

Consider the following graph where each node represents a DP state. An edge from node X to node Y (written as (X, Y)) denotes the recurrence relation that node Y depends upon node X's result for computation.



Changing direction of which edges would allow node 4 to use node 1's computation ?

A. (3, 4)
B. (2, 3)
C. (4, 5)
D. (1, 2)
E. (6, 7)

Question 4)

The King of St.Petersburg has come across a problem that he is confused about and now requires your help. St.Petersburg has a treasure that consists of N coins. The king now wants to find the largest subset of coins, say S, such that if coins A, B ϵ S then either A divides B or B divides A. You being his smartest ADA student are now required to find the right approaches to solve this task.

A. We can apply DP to solve the problem in O(N^4) by running 2 loops that denote the first and last indices respectively of the subset in the array of coins and then run 2 more loops that just checks whether each coin in this subset divides every other coin.
B. We can apply DP in O(N) by finding the longest subsequence S such that S[i] divides S[i+1].
C. We can apply DP to solve the problem in O(N^2) by sorting the numbers and then we need to find the longest subsequence S such that S[i] divides S[i+1].
D. DP is possible by requiring O(N * (2 ^ X)) time where X is the total number of subsets possible.
E. We cannot apply DP to this problem.

Question 5)

Given an integer valued array of size N, you were asked to find if there exists a subsequence having a sum S. You were aware of two algorithms,

1) Write down a DP solution, DP(n,s), which signifies that by using the first n elements of the array, can you generate a sum s.
2) generating all subsequences of the array using brute recursion, and individually checking if any of them sum to S.

Which of the following is/are true regarding the two algorithms you were aware of

a) Algorithm 1 is a more better version of Algorithm 2, and in no case would one prefer to use Algorithm 2 over algorithm 1
b) For a setup with N being constrained to a small value, Algorithm 2 may perform better than Algorithm 1
c) For a setup with S being constrained to a small value, there can arise a situation where Algorithm 2 performs better than Algorithm 1
d) For a setup with S being constrained to a small value, Algorithm 1 always performs better than Algorithm 2

Question 6)

Which of the following is the recurrence relation to get the minimum number of scalar multiplications to multiply the matrices give, where M[i-1] * M[i] gives the dimension of the ith matrix?

a) dp[i,j] = 1 if i=j
dp[i,j] = min{dp[i,k] + dp[k+1,j]}

b) dp[i,j] = 0 if i=j
dp[i,j] = min{dp[i,k] + dp[k+1,j]} for all k in i to j

c) dp[i,j] = 1 if i=j
dp[i,j] = min{dp[i,k] + dp[k+1,j]} + M[i-1]*M[k]*M[j].

d) dp[i,j] = 0 if i=j
dp[i,j] = min{dp[i,k] + dp[k+1,j]} + M[i-1]*M[k]*M[j].

Question 7)

You are given an n x n grid, where cell (i,j) is the cell of the i-th row and the j-th column, with (1,1) representing the top-left corner, (n,n) denoting the bottom right corner of the grid. Each cell (i,j) has an integer value a_{ij}. You want to map out a path from top-left to bottom right such that the values of the cells you traverse have the maximum sum possible. However, you can only move down or right one cell at every step. Which of the following recurrences represents a feasible way of doing the same?

(Where dp[i, j] is a certain dynamic programming function for the same)

a) dp[i, j] = a_{11} if i = j = 1
= dp[i-1,j] + a_{ij} if i > 1, j = 1
= dp[i, j-1] + a_{ij} if i = 1, j > 1
= max{dp[i-1,j], dp[i, j-1]} + a_{ij} otherwise
b) dp[i, j] = a_{nn} if i = j = n
= dp[i+1,j] + a_{ij} if i < n, j = n
= dp[i, j+1] + a_{ij} if i = n, j < n
= max{dp[i+1,j], dp[i, j+1]} + a_{ij} otherwise
c) Both are feasible recurrences
d) Neither is a feasible recurrence

Question 8)

Given an array of *n* weights $W = [w_1, w_2, \ldots, w_n]$, you need to select a subset W' of weights from this such that the sum of weights in W' is maximum, but no two adjacent from W are in W' (that is, both w_i and w_{i+1} are not in W' for any i ≥ 1). An intuitive approach is to check each weight w from W, starting from the heaviest, and checking them in non-increasing order of weights. If the

adjacent weights of w are not in W', w is added to W', else it is not. Is this approach correct? If not, select an input from which this will fail.

a) W = [1, 2, 3, 4, 5]
b) W = [2, 5, 6, 4, 3, 5]
c) W = [1, 5, 2, 4, 3, 6]
d) The proposed algorithm is correct and works correctly for all inputs

Question 9)

Which of the following does not use memoization?

a) Finding the n-th Fibonacci number by iteratively storing the results of all calculations from fib(1) to fib(n)
b) Using dynamic programming to solve the longest common subsequence problem
c) Finding the minimum operations required for matrix chain multiplication in $O(n^3)$ time
d) Printing the steps required to solve the Tower of Hanoi problem using recursive calls

Question 10)

Given an optimal substructure for a Dynamic Programming problem as OPT[a][b][c] = F(a,b,c) where F is some recurrence function in a,b,c. Further it is given to you that OPT[a][b][c] can always be evaluated. Which of the following is/are true about this Dynamic Programming problem.

a) Evaluating OPT[a][b][c] will take O(abc) time to be evaluated
b) Evaluating OPT[a][b][c] may or may not take O(abc) time to be evaluated
c) Given OPT[a][b][c] takes O(abc) time to be evaluated. You were somehow able to prove that c is a linear combination of a and b, then OPT[a][b][c] will take O(ab) to be evaluated.
d) Given OPT[a][b][c] takes O(abc) time to be evaluated. You were somehow able to prove that c is a linear combination of a and b, then OPT[a][b][c] may or may not take O(ab) to be evaluated.

—----------**END**—----------

ROUGH

Analysis and Design of Algorithms Rubrics
Quiz-2
Set-2

Date: 23.02.2023 **Duration: 75 minutes**

Question 1)

Given an integer valued array of size N, you were asked to find if there exists a subsequence having a sum S. You were aware of two algorithms,

1) Write down a DP solution, DP(n,s), which signifies that by using the first n elements of the array, can you generate a sum s.
2) generating all subsequences of the array using brute recursion, and individually checking if any of them sum to S.

Which of the following is/are true regarding the two algorithms you were aware of

a) Algorithm 1 is a more better version of Algorithm 2, and in no case would one prefer to use Algorithm 2 over algorithm 1
b) For a setup with N being constrained to a small value, Algorithm 2 may perform better than Algorithm 1
c) For a setup with S being constrained to a small value, there can arise a situation where Algorithm 2 performs better than Algorithm 1
d) For a setup with S being constrained to a small value, Algorithm 1 always performs better than Algorithm 2

Question 2)

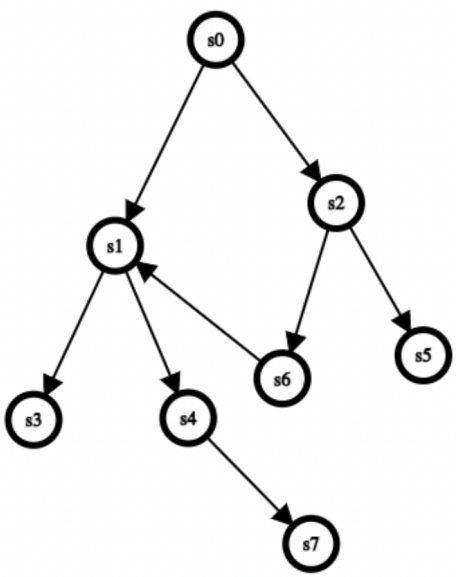
Given an array of *n* weights $W = [w_1, w_2, \ldots, w_n]$, you need to select a subset W' of weights from this such that the sum of weights in W' is maximum, but no two adjacent from W are in W' (that is, both w_i and w_{i+1} are not in W' for any $i \geq 1$). An intuitive approach is to check each weight w from W, starting from the heaviest, and checking them in non-increasing order of weights. If the

adjacent weights of w are not in W', w is added to W', else it is not. Is this approach correct? If not, select an input from which this will fail.

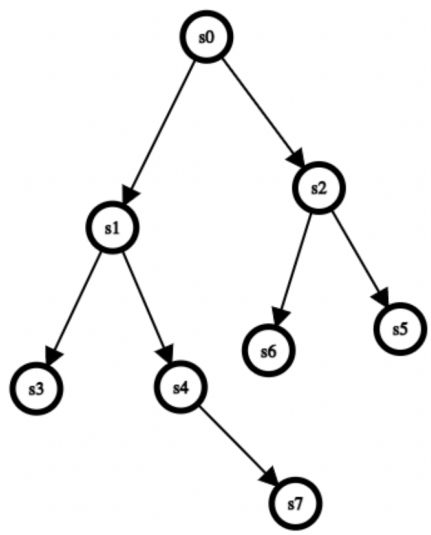
a) W = [1, 2, 3, 4, 5]
b) W = [2, 5, 6, 4, 3, 5]
c) W = [1, 5, 2, 4, 3, 6]
d) The proposed algorithm is correct and works correctly for all inputs

Question 3)

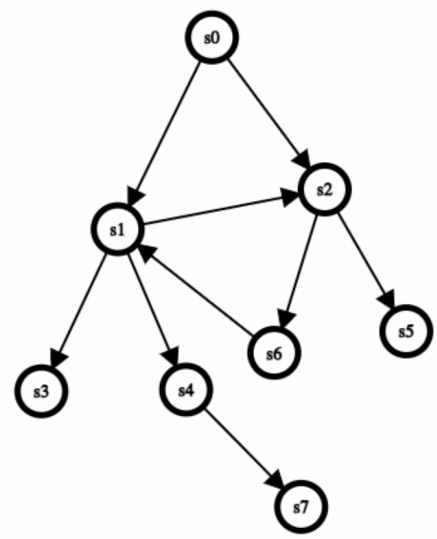
Recurrence 1



Recurrence 2



Recurrence 3



Given 3 Recurrences, to evaluate the s0 for each of the cases **optimally** which of the following is correct

a) 1->Recursion, 2->Dynamic Programming, 3->Recursion
b) 1->Can't be Evaluated using Rec or DP, 2->DP, 3->DP
c) 1->DP, 2->Recursion, 3->Recursion
d) 1->DP, 2->Recursion, 3->Can't be Evaluated using Rec or DP

Question 4)

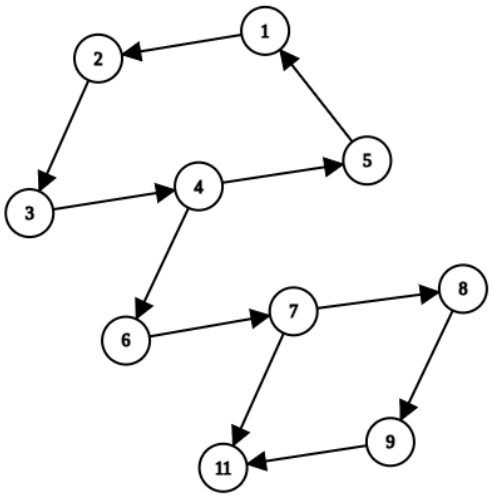
Consider a country whose coins are minted with denominations of {d1, . . . , dk } units. We want to count how many distinct ways C(n) there are to make change of n units. For example, in a

country whose denominations are {1, 6, 10}, C(5) = 1, C(6) to C(9) = 2, C(10) = 3, and C(12) = 4. Now, How many ways are there to change 20 units from {1, 6, 10}?

A. 20
B. 1
C. 7
D. 11
E. 10

Question 5)

Consider the following graph where each node represents a DP state. An edge from node X to node Y (written as (X, Y)) denotes the recurrence relation that node Y depends upon node X's result for computation.



Changing direction of which edges would allow node 4 to use node 1's computation ?

A. (3, 4)
B. (2, 3)
C. (4, 5)
D. (1, 2)
E. (6, 7)

Question 6)

The King of St.Petersburg has come across a problem that he is confused about and now requires your help. St.Petersburg has a treasure that consists of N coins. The king now wants to find the largest subset of coins, say S, such that if coins A, B ϵ S then either A divides B or B

divides A. You being his smartest ADA student are now required to find the right approaches to solve this task.

A. We can apply DP to solve the problem in O(N^4) by running 2 loops that denote the first and last indices respectively of the subset in the array of coins and then run 2 more loops that just checks whether each coin in this subset divides every other coin.
B. We can apply DP in O(N) by finding the longest subsequence S such that S[i] divides S[i+1].
C. We can apply DP to solve the problem in O(N^2) by sorting the numbers and then we need to find the longest subsequence S such that S[i] divides S[i+1].
D. DP is possible by requiring O(N * (2 ^ X)) time where X is the total number of subsets possible.
E. We cannot apply DP to this problem.

Question 7)

Returning ADA legend, Mr Kong the monkey is looking at a tree (tree here refers to trees in CS and not the Plant Kingdom). Each vertex of the tree contains some amount of bananas. Mr Kong being lazy wants you to find a single simple path in the tree which contains the most amount of bananas. A simple path in a tree is a path from vertex u to vertex v, such that no vertex in the path from u to v is repeated. Which of the following is/are true.

a) The vertex with the largest amount of bananas will always be in the optimal simple path
b) This can be solved using the following strategy: start from the vertex containing the maximum number of bananas, next go to the children of that vertex which has the maximum number of bananas, continue till you reach a leaf node.
c) If a dynamic programming approach was to be taken,
defined as: opt[v] = bananas[v] + max_over_all(opt[child[v]]), where opt[v] represents the optimal path including vertex v, is a correct recurrence, final answer will be in max(opt[v]) for all v in the tree.
d) Keeping track of the optimal at a vertex (opt[v]) is not enough in the recurrence given in option c and more information is needed.

Question 8)

Which of the following is the recurrence relation to get the minimum number of scalar multiplications to multiply the matrices give, where M[i-1] * M[i] gives the dimension of the ith matrix?

a) dp[i,j] = 1 if i=j
dp[i,j] = min{dp[i,k] + dp[k+1,j]}

b) dp[i,j] = 0 if i=j
dp[i,j] = min{dp[i,k] + dp[k+1,j]} for all k in i to j

c) dp[i,j] = 1 if i=j
dp[i,j] = min{dp[i,k] + dp[k+1,j]} + M[i-1]*M[k]*M[j].

d) dp[i,j] = 0 if i=j
dp[i,j] = min{dp[i,k] + dp[k+1,j]} + M[i-1]*M[k]*M[j].

Question 9)

You are given an n x n grid, where cell (i,j) is the cell of the i-th row and the j-th column, with (1,1) representing the top-left corner, (n,n) denoting the bottom right corner of the grid. Each cell (i,j) has an integer value a_{ij}. You want to map out a path from top-left to bottom right such that the values of the cells you traverse have the maximum sum possible. However, you can only move down or right one cell at every step. Which of the following recurrences represents a feasible way of doing the same?

(Where dp[i, j] is a certain dynamic programming function for the same)

a) dp[i, j] = a_{11} if i = j = 1
= dp[i-1,j] + a_{ij} if i > 1, j = 1
= dp[i, j-1] + a_{ij} if i = 1, j > 1
= max{dp[i-1,j], dp[i, j-1]} + a_{ij} otherwise

b) dp[i, j] = a_{nn} if i = j = n
= dp[i+1,j] + a_{ij} if i < n, j = n
= dp[i, j+1] + a_{ij} if i = n, j < n
= max{dp[i+1,j], dp[i, j+1]} + a_{ij} otherwise

c) Both are feasible recurrences

d) Neither is a feasible recurrence

Question 10)

Given an optimal substructure for a Dynamic Programming problem as OPT[a][b][c] = F(a,b,c) where F is some recurrence function in a,b,c. Further it is given to you that OPT[a][b][c] can

always be evaluated. Which of the following is/are true about this Dynamic Programming problem.

a) Evaluating OPT[a][b][c] will take O(abc) time to be evaluated
b) Evaluating OPT[a][b][c] may or may not take O(abc) time to be evaluated
c) Given OPT[a][b][c] takes O(abc) time to be evaluated. You were somehow able to prove that c is a linear combination of a and b, then OPT[a][b][c] will take O(ab) to be evaluated.
d) Given OPT[a][b][c] takes O(abc) time to be evaluated. You were somehow able to prove that c is a linear combination of a and b, then OPT[a][b][c] may or may not take O(ab) to be evaluated.

—----------**END**—----------

ROUGH

Analysis and Design of Algorithms Rubrics
Quiz-2
Set-3

Date: 23.02.2023

Question 1)

You are given an n x n grid, where cell (i,j) is the cell of the i-th row and the j-th column, with (1,1) representing the top-left corner, (n,n) denoting the bottom right corner of the grid. Each cell (i,j) has an integer value a_{ij}. You want to map out a path from top-left to bottom right such that the values of the cells you traverse have the maximum sum possible. However, you can only move down or right one cell at every step. Which of the following recurrences represents a feasible way of doing the same?

(Where dp[i, j] is a certain dynamic programming function for the same)

a) dp[i, j] = a_{11} if i = j = 1
   = dp[i-1,j] + a_{ij} if i > 1, j = 1
   = dp[i, j-1] + a_{ij} if i = 1, j > 1
   = max{dp[i-1,j], dp[i, j-1]} + a_{ij} otherwise
b) dp[i, j] = a_{nn} if i = j = n
   = dp[i+1,j] + a_{ij} if i < n, j = n
   = dp[i, j+1] + a_{ij} if i = n, j < n
   = max{dp[i+1,j], dp[i, j+1]} + a_{ij} otherwise
c) Both are feasible recurrences
d) Neither is a feasible recurrence

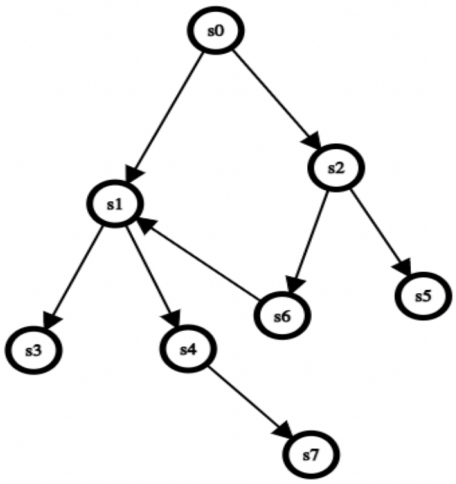
Question 2)

Given an array of *n* weights $W = [w_1, w_2, \ldots, w_n]$, you need to select a subset W' of weights from this such that the sum of weights in W' is maximum, but no two adjacent from W are in W' (that is, both w_i and w_{i+1} are not in W' for any i ≥ 1). An intuitive approach is to check each weight w from W, starting from the heaviest, and checking them in non-increasing order of weights. If the adjacent weights of w are not in W', w is added to W', else it is not. Is this approach correct? If not, select an input from which this will fail.

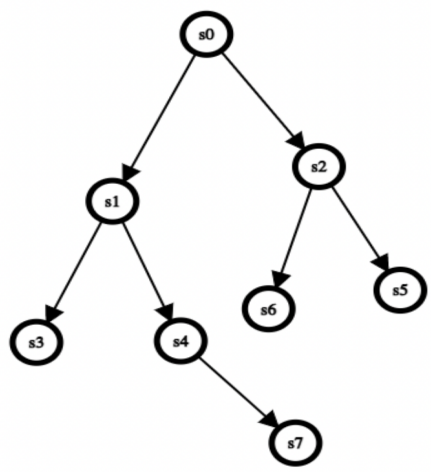
a) W = [1, 2, 3, 4, 5]
b) W = [2, 5, 6, 4, 3, 5]
c) W = [1, 5, 2, 4, 3, 6]
d) The proposed algorithm is correct and works correctly for all inputs

Question 3)

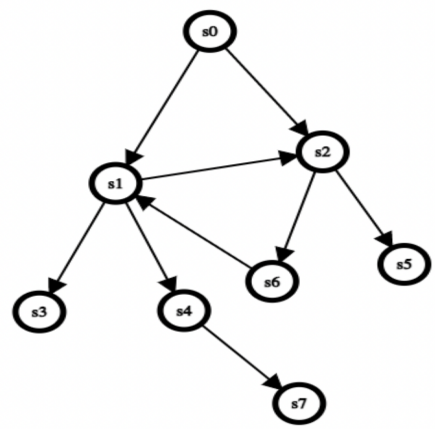
Recurrence 1



Recurrence 2



Recurrence 3



Given 3 Recurrences, to evaluate the s0 for each of the cases **optimally** which of the following is correct

a) 1->Recursion, 2->Dynamic Programming, 3->Recursion
b) 1->Can't be Evaluated using Rec or DP, 2->DP, 3->DP
c) 1->DP, 2->Recursion, 3->Recursion
d) 1->DP, 2->Recursion, 3->Can't be Evaluated using Rec or DP

Question 4)

Consider a country whose coins are minted with denominations of {d1, . . . , dk } units. We want to count how many distinct ways C(n) there are to make change of n units. For example, in a country whose denominations are {1, 6, 10}, C(5) = 1, C(6) to C(9) = 2, C(10) = 3, and C(12) = 4. Now, How many ways are there to change 20 units from {1, 6, 10}?

A. 20 B. 1 C. 7 D. 11 E. 10

Question 5)

Given an integer valued array of size N, you were asked to find if there exists a subsequence having a sum S. You were aware of two algorithms,
1) Write down a DP solution, DP(n,s), which signifies that by using the first n elements of the array, can you generate a sum s.
2) generating all subsequences of the array using brute recursion, and individually checking if any of them sum to S.

Which of the following is/are true regarding the two algorithms you were aware of

a) Algorithm 1 is a more better version of Algorithm 2, and in no case would one prefer to use Algorithm 2 over algorithm 1
b) For a setup with N being constrained to a small value, Algorithm 2 may perform better than Algorithm 1
c) For a setup with S being constrained to a small value, there can arise a situation where Algorithm 2 performs better than Algorithm 1
d) For a setup with S being constrained to a small value, Algorithm 1 always performs better than Algorithm 2

Question 6)
Which of the following is the recurrence relation to get the minimum number of scalar multiplications to multiply the matrices give, where M[i-1] * M[i] gives the dimension of the ith matrix?

a) dp[i,j] = 1 if i=j
   dp[i,j] = min{dp[i,k] + dp[k+1,j]}

b) dp[i,j] = 0 if i=j
   dp[i,j] = min{dp[i,k] + dp[k+1,j]} for all k in i to j

c) dp[i,j] = 1 if i=j
   dp[i,j] = min{dp[i,k] + dp[k+1,j]} + M[i-1]*M[k]*M[j].

d) dp[i,j] = 0 if i=j
   dp[i,j] = min{dp[i,k] + dp[k+1,j]} + M[i-1]*M[k]*M[j].

Question 7)

The King of St.Petersburg has come across a problem that he is confused about and now requires your help. St.Petersburg has a treasure that consists of N coins. The king now wants to find the largest subset of coins, say S, such that if coins A, B ϵ S then either A divides B or B divides A. You being his smartest ADA student are now required to find the right approaches to solve this task.

A. We can apply DP to solve the problem in O(N^4) by running 2 loops that denote the first and last indices respectively of the subset in the array of coins and then run 2 more loops that just checks whether each coin in this subset divides every other coin.
B. We can apply DP in O(N) by finding the longest subsequence S such that S[i] divides S[i+1].
C. We can apply DP to solve the problem in O(N^2) by sorting the numbers and then we need to find the longest subsequence S such that S[i] divides S[i+1].
D. DP is possible by requiring O(N * (2 ^ X)) time where X is the total number of subsets possible.
E. We cannot apply DP to this problem.

Question 8)

Returning ADA legend, Mr Kong the monkey is looking at a tree (tree here refers to trees in CS and not the Plant Kingdom). Each vertex of the tree contains some amount of bananas. Mr Kong being lazy wants you to find a single simple path in the tree which contains the most amount of bananas. A simple path in a tree is a path from vertex u to vertex v, such that no vertex in the path from u to v is repeated. Which of the following is/are true.

a) The vertex with the largest amount of bananas will always be in the optimal simple path
b) This can be solved using the following strategy: start from the vertex containing the maximum number of bananas, next go to the children of that vertex which has the maximum number of bananas, continue till you reach a leaf node.
c) If a dynamic programming approach was to be taken,
defined as: opt[v] = bananas[v] + max_over_all(opt[child[v]]), where opt[v] represents the optimal path including vertex v, is a correct recurrence, final answer will be in max(opt[v]) for all v in the tree.
d) Keeping track of the optimal at a vertex (opt[v]) is not enough in the recurrence given in option c and more information is needed.

Question 9)

Given an optimal substructure for a Dynamic Programming problem as OPT[a][b][c] = F(a,b,c) where F is some recurrence function in a,b,c. Further it is given to you that OPT[a][b][c] can always be evaluated. Which of the following is/are true about this Dynamic Programming problem.

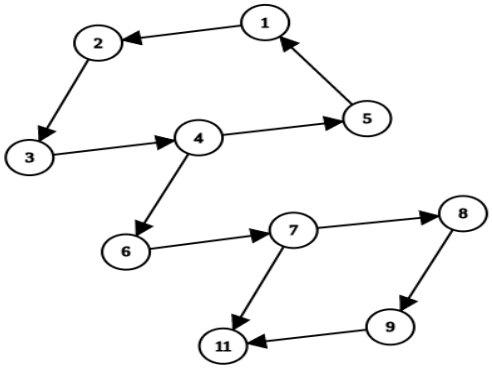
a) Evaluating OPT[a][b][c] will take O(abc) time to be evaluated
b) Evaluating OPT[a][b][c] may or may not take O(abc) time to be evaluated

c) Given OPT[a][b][c] takes O(abc) time to be evaluated. You were somehow able to prove that c is a linear combination of a and b, then OPT[a][b][c] will take O(ab) to be evaluated.

d) Given OPT[a][b][c] takes O(abc) time to be evaluated. You were somehow able to prove that c is a linear combination of a and b, then OPT[a][b][c] may or may not take O(ab) to be evaluated.

Question 10)

Consider the following graph where each node represents a DP state. An edge from node X to node Y (written as (X, Y)) denotes the recurrence relation that node Y depends upon node X's result for computation.



Changing direction of which edges would allow node 4 to use node 1's computation ?

A. (3, 4)
B. (2, 3)
C. (4, 5)
D. (1, 2)
E.(6, 7)

—----------**END**—----------

ROUGH

Quiz-3 -ADA Rubrics

Q1. Let G be a simple undirected graph. Let T_d be a depth first search tree of G and T_b be a breadth first search tree of G.
Consider the following statements.
(I) For every edge (u,v) of G, if u is at depth i and v is at depth j in T_b, then |i-jl=1.
(II) No edge of G is a cross edge with respect to T_d.
Which of the statements above must necessarily be true?

A. I only
B. II only
C. Both I and II
D. Neither I or II

Q2. Given the frequency of letters in a string as: B-1, C-6, A-5, D-3. Find the huffman code for character D.

A. 11
B. 100
C. 0
D. 101

Q3. Let G be an undirected graph. Consider a depth-first traversal of G, where T is the depth-first search tree that results. Let u be the first new (unvisited) vertex visited in the traversal, and v be its first new (unvisited) vertex visited after u. Which of the assertions below is always true?

(A) In G, u,v must be an edge, while in T, u is a descendent of v.
(B) In G, u,v must be an edge, while in T, v is an ancestor of u.
(C) If u,v in G is not an edge, then u in T is a leaf
(D) If u,v in G is not an edge, then u and v in T must have the same parent.

Q4. In Huffman Coding the code length does not depend on the frequency of occurrence of characters.

a) True
b) False

Q5. Which of the following is/are the applications of MST ?

i. Finding diameter of the graph
ii. Finding bipartite graph
iii. Telephone networks
iv. Sudoku solver

A. i and ii
B. iii
C. iii and iv
D. ii and iv

Q6. Breadth First Search(BFS) is started on a binary tree beginning from the root vertex. There is a vertex t at a distance five from the root. If t is the n-th vertex in this BFS traversal, then the maximum possible value of n is _____. (Consider root to be at distance 0)

A. 63
B. 15
C. 31
D. 64

Q7. You are given a directed graph G = (V,E) with edge lengths, and a source vertex s. However it is given that all the negative edges are outgoing from s.
Say there is also no negative cycle. Then, running Djikstra from s -

A. Will always find the shortest path from s to all vertices reachable from s
B. Will find the shortest path from s to neighbors of s but not necessarily to other vertices
C. Will find the shortest path from s to itself but not necessarily to other vertices
D. None of the above

Q8. You are given an undirected graph G = (V,E) with positive integer distinct edge lengths. Consider a minimum spanning tree (MST) T of G. Consider the shortest path P from a source vertex s to a destination vertex v. Now, suppose we decrease the length of every edge by 1. Then-

A. T necessarily continues to be an MST and P necessarily continues to be a shortest path from s to v
B. T necessarily continues to be an MST and P is not necessarily now a shortest path
C. T is not necessarily now an MST and P necessarily continues to be a shortest path
D. None of the above

Q9. Read the following statements -
I) Strongly connected components are not affected by reversal of all edges in terms of vertex reachability.
II) Strongly connected components are not affected by reversal of all edges in terms of vertices reachability.
Which of the following statements is/are **correct**?

A. I only
B. II only
C. Both I and II
D. Neither I or II

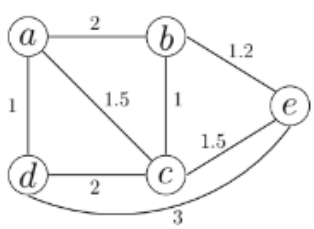
Q10. Read the following statements -
I) A DFS of a directed graph generally produces the exact number of edges of a tree, i.e., not dependent on the order in which vertices are considered for DFS.
II) If the finishing time f[u] > f[v] of DFS for two vertices u and v in a graph G which is directed, and u and v are in the DFS tree same as each other in the DFS forest, then u is an ancestor of v in the depth-first tree.
Which of the following statements is/are **incorrect**?

A. I only
B. II only
C. Both I and II
D. Neither I or II

Q11. Read the following statements -
I) Consider an edge weighted undirected connected graph G such that all edge weights are positive and distinct. Let e* be the edge with smallest weight in G. Then, G has a minimum spanning tree that does not contain the edge e*.
II) Let G be a directed graph having s, t ∈ V (G) and positive edge weights. Consider a longest simple path P (i.e. a path P with maximum total edge weight) from s to t in G. Let u ∈ P be an internal vertex in this path P. Then, the subpath of P from s to u is the longest simple path from s to u in G.
Which of the following statements is/are **correct**?

A. I only
B. II only
C. Both I and II
D. Neither I or II

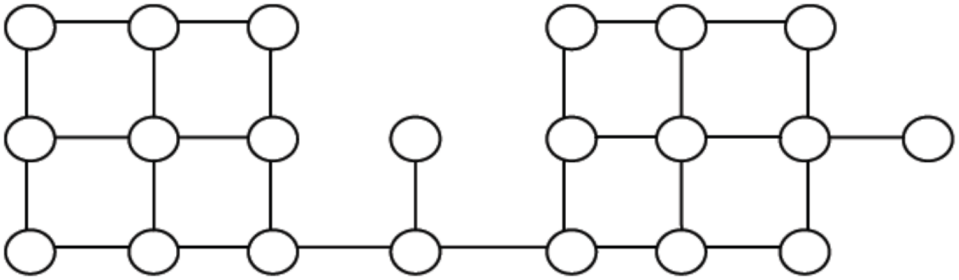
Q12. Consider the following undirected graph G



Which of the following edge(s) is/are necessarily in every MST of G ?

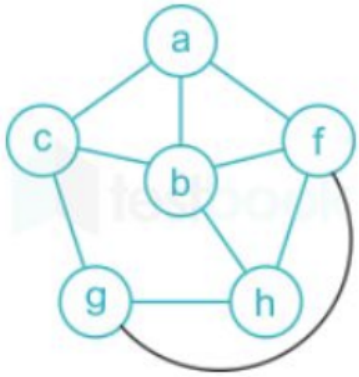
A. ac, be
B. ad, ce
C. bc
D. dc, eb

Q13. Suppose depth first search is executed on the graph below starting at some unknown vertex. Assume that a recursive call to visit a vertex is made only after first checking that the vertex has not been visited earlier. Then the maximum possible recursion depth (including the initial call) is _____.



A. 16
B. 19
C. 17
D. 20

Q14. Consider the following graph:



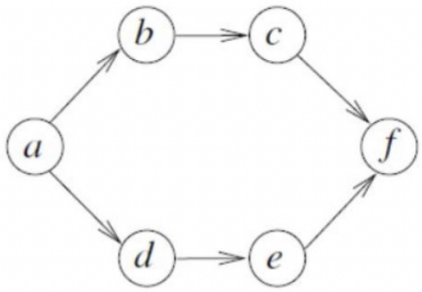
For the above graph, the following sequences of DFS are given -

(A) abcghf
(B) abfchg
(C) abfhgc
(D) afghbc

Which of the following is correct ?

A. (A), (B), (D)
B. (A), (C), (D)
C. (B), (C), (D)
D. (A), (B), (C), (D)

Q15. Consider the following graph -



The number of different topological orderings of the vertices of the graph is:

A. 4
B. 5
C. 6
D. 7

Quiz4 (ADA-2023)
Answer keys

Q1. Given a graph G = (V, E) with positive edge weights, the Bellman-Ford algorithm and Dijkstra's algorithm can produce different shortest-path trees despite always producing the same shortest-path weights. True or False ?

Solution: True.

Q2. Dijkstra's Algorithm has lesser time complexity compared to Bellman Ford's algorithm, not considering the space complexity.

a) True
b) False

Q3. Given G= (N, M) where N is the number of vertices and M is the number of edges, it is known that G is a complete undirected graph. What is the total number of **spanning trees** for G ?

a. N * 2
b. N ^ 2
c. N ^ (N - 2)
d. N ^ (M - 1)

Q4. Read the following statements -
I) Let G = (V,E) be a flow network, with a source s, a sink t, and a positive integer capacity c on every edge e. If f is a maximum (s, t)-flow in G, then for all e ∈ E that goes out of s, f(e) = c.
II) Let G = (V,E) be a flow network with source s, sink t and all the edges of G have odd capacities. Then for any maximum (s, t)-flow f and for any edge e ∈ E, f(e) is either zero or odd.
Which of the following statements is/are **incorrect**?

A. I only
B. II only
C. Both I and II
D. Neither I or II

Q5. Read the following statements -

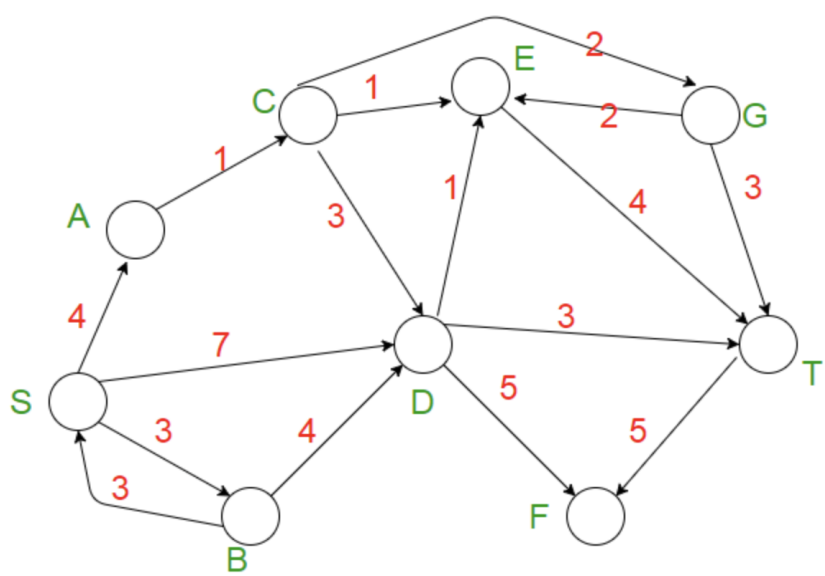
I) Let G be a directed graph with positive edge weights w : E(G) → R.
Suppose that we modify the graph G into G' as follows. For every edge e ∈ E(G), we set w'(e) = w(e)/2 to be the modified weights in G'. Then, every shortest path from s to t in G is a shortest path from s to t in G'.

II) Let G = (V,E) be a flow network, with a source s, a sink t, and a positive capacity c on every edge e. Suppose (A, B) is the unique s-t minimum cut in G w.r.t these capacities. Consider a modified graph G' where the capacity of every edge is increased by 1. Then (A, B) is still a minimum (s, t)-cut for G'.

Which of the following statements is/are correct?

A. I only
B. II only
C. Both I and II
D. Neither I or II

Q6. Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a vertex v is updated only when a strictly shorter path to v is discovered.



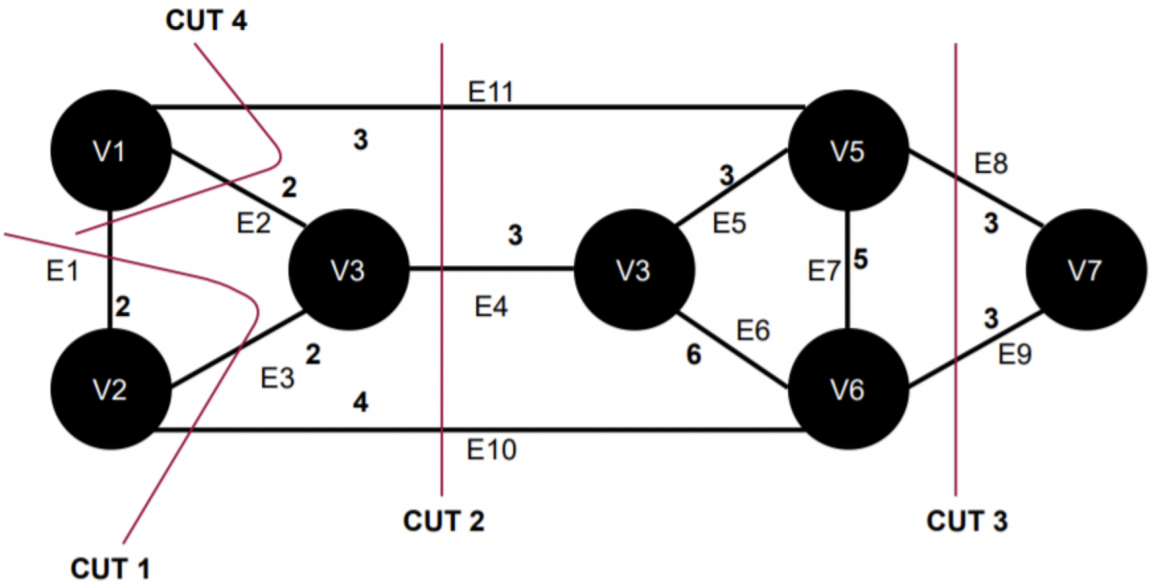
A. SDT
B. SBDT

C. SACDT
D. SACET

Q7. Consider the following statements given below:
S1 : If a graph contains a negative weight cycle then Dijkstra's algorithm may or may not terminate.
S2 : The Bellman-Ford algorithm guarantees that it will always produce a shortest path between two given vertices u and v in any weighted graph.
Which of the above statements are **incorrect**?

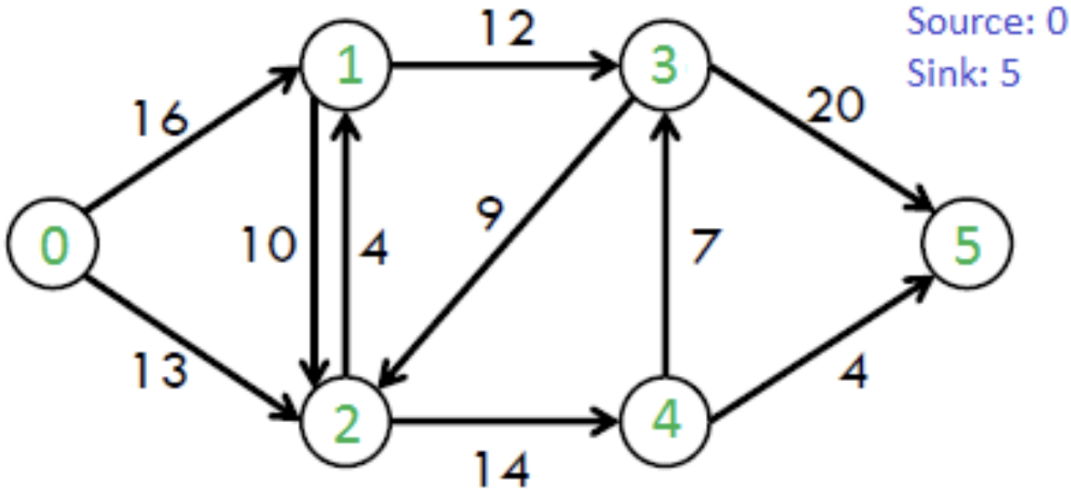
1. Only S1
2. Only S2
3. Both S1 and S2
4. None of these

Q8. Which cut is the minimum cut of the following network?



A. CUT 3
B. CUT 1
C. CUT 2
D. CUT 4
E. None of the above

Q9. The value of max-flow in the following graph is:



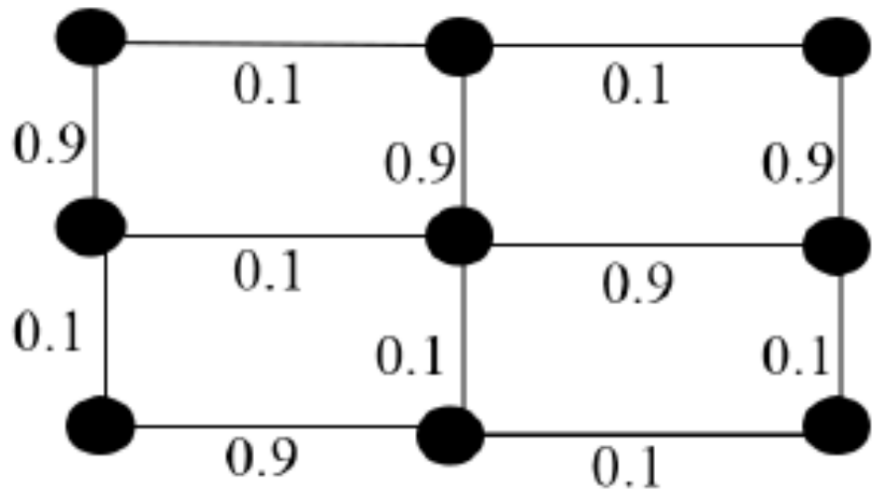
A. 22
B. 23
C. 24
D. 25

Q10. Suppose that f is a feasible but not maximum s, t-flow in a network with nonnegative capacity. Read the statements below and choose the correct statement.

S1. Residual graph Gf will never have a path from s to t that was not a path from s to t in the original graph G.
S2. The residual graph Gf can have a path P∗ from s to t with min{capacity(e) : e ∈P∗} > 0 that is not there at all in the original graph.
S3. The residual graph Gf will always have a path P^* from s to t with min{capacity(e) :e∈P^*} that is not there at all in the original graph.
S4. None of the above

A. S1
B. S2
C. S3
D. S4

Q11. Consider the following undirected graph with edge weights as shown:



The total number of possible minimum spanning trees are-

A. 2
B. 3
C. 4
D. 6

Q12. Read the following statements:
I) In a flow network, the maximum amount of flow passing from the source to the sink is equal to the total weight of the edges in a minimum cut.
II) Let f be a flow on a network (G,c) with net flow v and let C be a vertex cut (S, T) with capacity k. Then v > k.
Which of the above statements is/are correct:

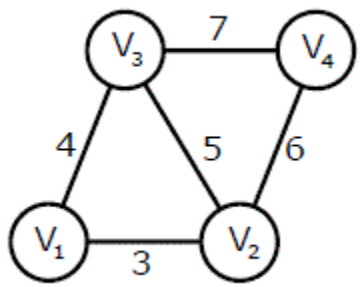
A. Only I
B. Only II
C. Both I and II
D. Neither I nor II

Q13. Each maximum flow defines a minimum capacity cut, using the method discussed in class. Claim: This minimum capacity cut is unique, i.e., for a given maximum flow, there could not be more than one minimum capacity cut for this flow.

A. True

B. False

Q14. An undirected graph G(V, E) contains n (n > 2) nodes named v1 , v2 ,....vn. Two nodes vi , vj are connected if and only if 0 < |i – j| <= 2. Each edge (vi, vj) is assigned a weight i + j. A sample graph with n = 4 is shown below. What will be the cost of the minimum spanning tree (MST) of such a graph with n nodes?



a. (11n^2 – 5n) / 12
b. n^2 – n + 1
c. 6n – 11
d. 2n + 1

Q15. Let G be an undirected connected graph with distinct edge weight. Let e_{max} be the edge with maximum weight and e_{min} the edge with minimum weight. Which of the following statements is false?

A. Every minimum spanning tree of G must contain e_{min}
B. If e_{max} is in a minimum spanning tree, then its removal must disconnect G
C. No minimum spanning tree contains e_{max}
D. G has a unique minimum spanning tree