Algorithms CS2500

Dr. Simone Silvestri
Compiled by Ken Goss

April 18, 2016

Part I
Introduction to Algorithms

1 Comparing Algorithms

Definition: Given n numbers $< a_1$, $a_n >$ find a permutation of these numbers such that the numbers are in increasing order

- Solutions:

Insertion Sort - $C_1(n^2)$	Merge Sort - $C_2 n \, log_2 n$

Compare these Algorithms with two computers

A	B
Intel Core i7	Intel 386 from about 1985
$\approx 10^{11}$Instructions per second	$\approx 10^7$Instructions per second
A is a better programmer than B so $C_1 < C_2$: C_1= 2 & C_2= 50	
Execution of each algorithm will be on a list containing 10^8 numbers	
Insertion Sort - $C_1(n^2)$	Merge Sort - $C_2 n \, log_2 n$
A = $\left(\frac{2(10^8)^2}{10^{11}}\right)$=2 $\cdot 10^5 s$	B = $\left(\frac{50 \cdot 10^8 \cdot log_2(10^8)}{10^7}\right)$=500 $\cdot log_2(10^8)$
$\approx$ 5.5 hours	$\approx$ 1.2 hours

2 Describing Algorithms

Pseudo Code conventions High level abstract language Unconcerned with details No need for type defininitions or instantiation

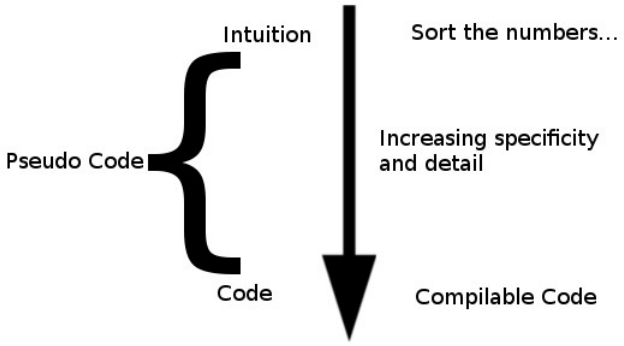


Figure 1: Pseudo Code gradient range

3 Insertion Sort

The Iterative Sorting Algorithm
Idea: an array containing all the numbers is traversed with an element in a placeholder till its appropriate location among the sorted numbers is found. This is repeated for each of the unsorted numbers until the array is completely sorted.

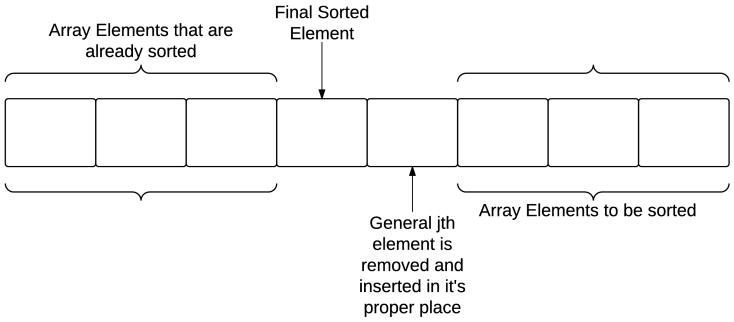


Figure 2: Insertion sort functional intuition

3.1 Pseudo Code for Insertion Sort

```
1  InsertionSort(A, n)begin
2      for j=2 to n do
3          key=A[j];
4          i=j-1;
5          while A[i] > key and i > 0 do
6              A[i+1] = A[i];
7              i = i-1;
8          end
9          A[i+1] = key;
10     end
11 end
```

Algorithm 1: Insertion Sort

3.2 Example: Sort $< 52143 >$

First for iteration: j=2 key=2 i=1
A[i] > key $\rightarrow <$ 55143 >
i $\not>$ 0 $\rightarrow$ exit while loop
A[i] = key < 25143 >

Second for iteration: j=3 key=1 i=2
A[i] >key $\rightarrow <$ 25543 >
i >0 and A[i]> key $\rightarrow <$ 22543 >
i $\not>$0 exit while$\rightarrow <$ 12543 >

Third for iteration j=4 key=4 i=3
A[i] > key $\rightarrow <$ 12553 >
A[i] $\not>$key exit while $\rightarrow <$ 12453 >

Fourth and final for iteration j=5 key=3 i=4
A[i] > key $\rightarrow <$ 12455 >
i > 0 and A[i] > key $\rightarrow <$ 12445 >
i > 0 but A[i] $\not>$key exit while $\rightarrow <$ 12345 >

4 Complexity

Understand the growth of the function's runtime with respect to the size of the arrray i.e. the number of items contained in the array to be sorted, $|n|$. Total runtime isthe sum of the executions of each of the instructions in the code. Each line (i) of code has a runtime c_i which is a known constant and t_j is the number of times that the below "while" loop is executed at the iteration j of the outer for loop.

The runtime T(n) is therefore, in this case, given my the following equation:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

In the best possible case, the array is already sorted which implies that $t_j = 1$,

Line Number	Pseudocode	Runtime	Number of Executions
	InsertionSort(A, n) {		
1	for j=2 to n {	c_1	n
2	key=A[j]	c_2	n-1
3	i=j-1	c_3	n-1
4	while(i>0 and A[i]>key){	c_4	$\sum_{j=2}^{n} t_j$
5	A[i+1]=A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
6	i=i-1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	}		
8	A[i+1]=key	c_7	n-1
9	}		

Figure 3: Algorithm Complexity Analysis

leading to the function operating in linear time, which yields the following:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (1-1) + c_6 \sum_{j=2}^{n} (1-1) + c_7(n-1)$$

$$\approx c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 n + c_6 n + c_7(n-1)$$

$$\approx (c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7)n \approx c_{net} \cdot n$$

In the worst possible case the array is sorted in reverse order i.e. ascending instead of descending order or vice-versa. This implies that $t_j = j$, leading to the function operating in quadratic time complexity, which yields the following:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j-1) + c_6 \sum_{j=2}^{n} (j-1) + c_7(n-1)$$

$$\approx c_1 n + c_2(n-1) + c_3(n-1) + c_4 \left(\frac{n(n-1)}{2} - 1 \right) + c_5 \left(\frac{n(n-1)}{2} \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7(n-1)$$

Following some algebraic manipulation, and combination of constants, the following form demonstrating the quadratic nature of this function's time complexity may be derived.

$$T(n) \approx an^2 + bn + c$$

4.1 Asymptotic Analysis

The goal is to compare the performance (runtime) of different algorithms without implementation details. Consider sizes of input which are "sufficiently large". In this situation, the constants are insignificant and therefore irrelevant due to the fact that the leading factor (the highest power of n) determines the complexity.

Big O Notation The object of this notation and analysis is the asymptotic upper bound of a function. Intuitively it may be thought of in terms of the idea that "The function can do no worse than..."

Definition 4.1 *Given a function g(n), and* $n \in \mathbb{N}$ *it is said that* $O(g(n)) = f(n)$ *if* $\exists(c > 0 \forall n_0 > 0) : 0 \leq f(n) \leq cg(n) \forall n \geq n_0$

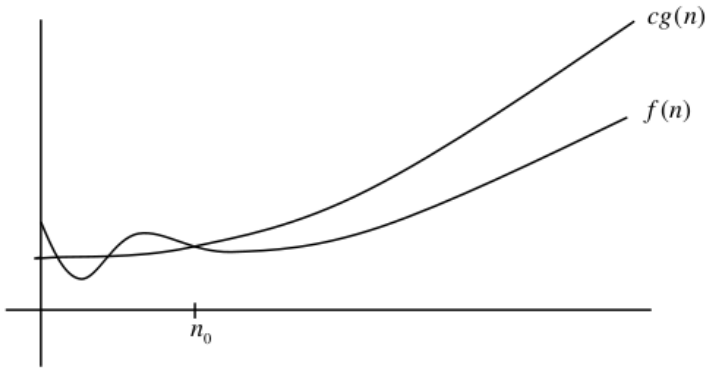


Figure 4: Big O relationship graph

Example 1: $f(n) = 3n + 3$ Prove $f(n) = O(n)$
How to solve? Find values of c and n_0 that satisfy the necessary conditions.

$$3n + 3 \leq cn \quad \forall \quad n \geq n_0$$

divide by n

$$3 + \frac{3}{n} \leq c$$

as n grows, and for all values > 3, $\frac{3}{n} \ll 1$

$$3 + small\# \leq c$$

4 is a good candidate - Don't make it more complicated than necessary
Solve the remaining inequality for n

$$3n + 3 \leq 4n$$

$$3 \leq n$$

Therefore $n_0 = 3$ and c=4 satisfies the conditions sought to be proven.

Example 2: $f(n) + 3n + 3$ Prove $f(n) = O(n^2)$

$$3n + 3 \leq cn^2$$

$$\frac{3}{n} + \frac{3}{n^2} \leq c \rightarrow c = 2$$

$$3n + 3 \leq 2n^2$$

$$2n^2 - 3n - 3 \geq 0$$

$$n_0 = \left\lceil \frac{3 \pm \sqrt{9 - 4 \cdot 2 \cdot -3}}{4} \right\rceil$$

Choose maximum (ceiling) value, so $n_0 \approx 5$

Big Ω is an asymptotic lower bound

Definition 4.2 *Given a function g(n), and* $n \in \mathbb{N}$ *it is said that* $\Omega(g(n)) = f(n)$ *if* $\exists(c > 0 \forall n_0 > 0) : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0$

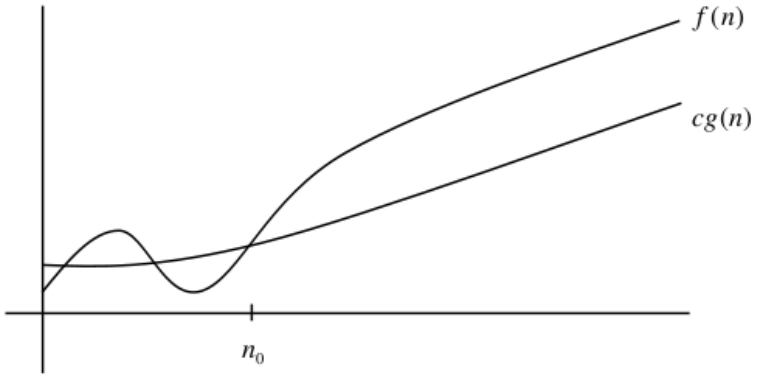


Figure 5: Big Ω relationship graph

Example 3: $f(n) = 2n^2 + 3$ Prove $f(n) = \Omega(\mathrm{n})$

$$2n^2 + 3 \geq cn \rightarrow c = 1$$

$$2n^2 + 3 \geq n$$

$$true \quad \forall n \geq 0$$

$$\therefore n_o = 0$$

Example 4: $f(n) = 2n^2 + 3$ Prove $f(n) = \Omega(n^2)$

$$2n^2 + 3 \geq cn \rightarrow c = 1$$

$$2n^2 + 3 \geq n$$

$$true \quad \forall n \geq 0$$

$$\therefore n_o = 0$$

Big Θ is an asymptotic tight bound

Definition 4.3 *Given a function g(n), and* $n \in \mathbb{N}$ *it is said that* $\Theta(g(n)) = f(n)$ *if* $\exists(c_1, c_2 > 0 \forall n_0 > 0) : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0$

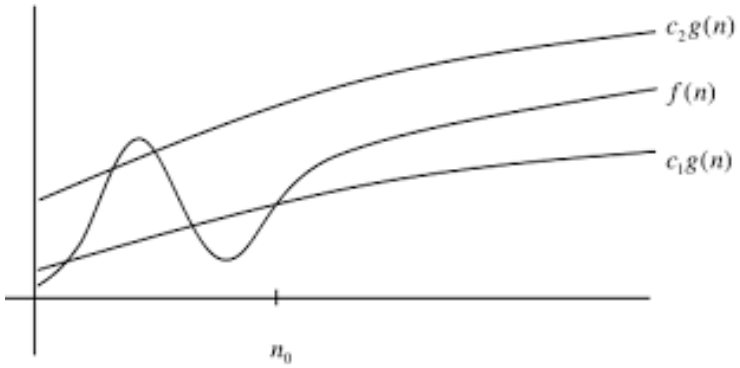


Figure 6: Big Θ relationship graph

Example 5: $f(n) = \frac{n^2}{2} - 3n$ Prove $f(n) = \Theta(n^2)$

$$c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

$$\forall n \geq 12 \rightarrow c_1 = \frac{1}{4}, c_2 = 1$$

$$\frac{1}{4}n^2 \leq \frac{n^2}{2} - 3n \leq n^2$$

$$true \quad \forall n \geq 12$$

$$\therefore n_o = 12, \quad c_1 = \frac{1}{4}, \quad c_2 = 1$$

4.2 Algebra of Asymptotic Notation

Definition 4.4 (Interrelationship of O Ω and Θ)
given f(n) and g(n)
f(n)=Θ(g(n)) iff f(n)=Ω(g(n)) and f(n)=O(g(n))

Definition 4.5 (Constants)
$\forall\ k \in \mathbb{R}^+$
if f(n)=O(g(n)) then kf(n)=O(g(n))
which implies constants have no effect
also applies to Θ as well as Ω

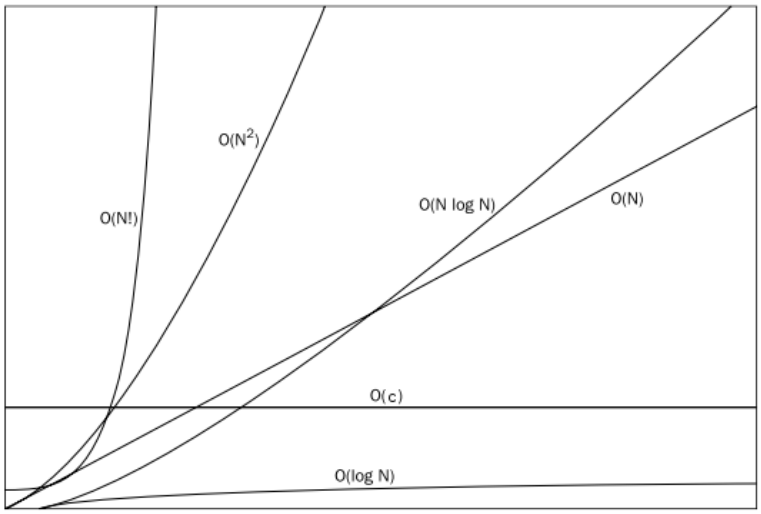


Figure 7: Common complexities graphed concurrently

Definition 4.6 (Sums)
if f(n)=O(g(n)) AND d(n)=O(h(n))
then f(n)+d(n)=O(g(n)+h(n))=O(max[g(n), h(n)])
also applies to Θ *as well as* Ω
Note: Hidden Constants - Algorithms exist that have better efficiency only at extreme limits (i.e. Counting Sort)

Definition 4.7 (Multiplication)
if f(n)=O(g(n)) AND d(n)=O(h(n))
then $f(n) \cdot d(n) = O(g(n) \cdot h(n))$
also applies to Θ *as well as* Ω

Definition 4.8 (Transitivity)
if f(n)=O(g(n)) AND g(n)=O(h(n))
then f(n)=O(h(n))
also applies to Θ *as well as* Ω

Definition 4.9 (Reflexivity)
f(n)=O(f(n))
also applies to Θ *as well as* Ω

Definition 4.10 (Symmetry)
iff $f(n)=\Theta(g(n))$ *then* $g(n)=\Theta(f(n))$
iff $f(n)=O(g(n))$ *then* $g(n)=\Omega(f(n))$
iff $f(n)=\Omega(g(n))$ *then* $g(n)=O(f(n))$

5 The Search Problem

Searching an ordered sequence: Given an ordered sequence $< a_1, a_2, ...a_n >$ and a value x, determine if x is in the sequence or not.
Trivial Solution: Iterate through the whole sequence: O(n)
Better Solution: Binary Search
Binary search is a recursive function in the base case, the array is of length 1, if that element is equal to x return True, else return False. Its recursive step is executed when the array is longer than a single element.

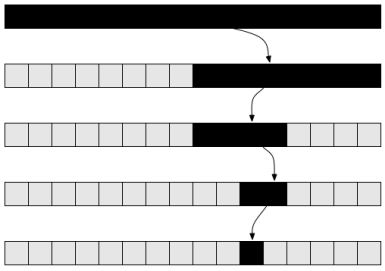


Figure 8: Binary Search Execution

5.1 Pseudo Code for Binary Search

```
1  BinarySearch(A, start, end)begin
2  |  //(A, 1, n) for first call
3  |  if start==end then
4  |  |  if A[start]==x then
5  |  |  |  return True
6  |  |  end
7  |  |  return False
8  |  end
9  |  m=⌈(start+end)/2⌉
10 |  if A[m] > x then
11 |  |  return BinarySearch(A, start, m-1)
12 |  end
13 |  return BinarySearch(A, m, end)
14 end
```

Algorithm 2: Binary Search

5.2 Example: Search $< 1, 3, 7, 9, 11 >$ for x=9

First call: start=1 end =5
start $\neq$ end $\rightarrow$ m=$\lceil \frac{5+1}{2} \rceil = 3 \rightarrow$ A[3]=7$\not>$x
recursive call BinarySearch(A, m, end)

Second call: start=3 end=5
start $\neq$ end $\rightarrow$ m=$\lceil \frac{5+3}{2} \rceil = 4 \rightarrow$ A[4]=9$\not>$x
recursive call BinarySearch(A, m, end)

Third call: start=4 end=5
start $\neq$ end $\rightarrow$ m=$\lceil \frac{5+4}{2} \rceil = 5 \rightarrow$ A[5]=11>x
recursive call BinarySearch(A, start, m-1)

Fourth call: start=4 end=4
start = end $\rightarrow$ A[4]=9==x
return True

6 Recurrence Equations

$$T(n) = \begin{cases} base \quad \Theta(1) & n = 1 \\ recursive \quad T(\frac{n}{2}) + \Theta(1) & n > 1 \end{cases}$$

Recursive Tree method:

- Nodes represent the cost of a single sub-problem at that level of recursion
- Each level of the tree is a level of recursion
- Between the root and leaves, but excluding the root and leaves, are intermediate recursive calls
- Leaves are the base cases
- Normally, write three specific levels then find the general level expression
- Unless otherwise specified all logarithms are assumed to be log_2

6.1 Example based on the above recurrence equation:

Tree	Level	Input Size	Cost
c	0	n	c
c	1	$\frac{n}{2}$	c
c	2	$\frac{n}{4}$	c
c	i	$\frac{n}{2^i}$	c
c	k (leaf or base case)	$\frac{n}{2^k} = 1$	c

Figure 9: Recursion Tree table basic example

Solve for k which represents the level at which the base case is reached:

$$1 = \frac{n}{2^k}$$

$$2^k = n$$

$$k = log_2(n)$$

Use this to find the total runtime:

$$\sum_{i=0}^{k-1} c + c = \sum_{i=0}^{log(n)-1} c + c = c \cdot log(n) + c = \Theta(log(n))$$

6.2 Example:

Recurrence Equation:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n^2) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

Tree	Level	Input Size	Cost
$c(n^2)$	0	n	$c(n^2)$
$c(n^2)$ $c(n^2)$	1	$\frac{n}{2}$	$\frac{c(n^2)}{2}$
$c(n^2)$ $c(n^2)$ $c(n^2)$ $c(n^2)$	2	$\frac{n}{4}$	$\frac{c(n^2)}{4}$
$c(n^2)$ ... $c(n^2)$	i	$\frac{n}{2^i}$	$\frac{c(n^2)}{2^i}$
$\Theta(1)$ ... $\Theta(1)$	k (leaf or base case)	$\frac{n}{2^k} = 1 \rightarrow k = log(n)$	$c \cdot 2^k \Theta(1)$

Figure 10: Recursion Tree table example

$$\sum_{i=0}^{log(n)-1} c + c = c\frac{n^2}{2^i} + cn = cn^2 \sum_{i=0}^{log(n)-1} (\frac{1}{2})^i + cn$$

use the summation expression property for geometric series

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \quad \forall x < 1$$

employed in our case:

$$cn^2 \sum_{i=0}^{log(n)-1} (\frac{1}{2})^i + cn = 2cn^2 + cn = \Theta(n^2)$$

7 The Master Theorem

(or, how to solve recurrence problems without trees)

Let $a \geq 1$, and $b \geq 1$ be constants and $f(n)$ be a function.
Consider $T(n) = aT(\frac{n}{b}) + f(n)$. This leads to three cases.

1. if $f(n) = O(n^{log_b(a)-\varepsilon})$ for some $\varepsilon > 0$
   then you may conclude that $T(n) = \Theta(n^{log_b(a)})$
2. if $f(n) = \Theta(n^{log_b(a)})$
   then you may conclude that $T(n) = \Theta(n^{log_b(a)} log(n))$
3. if $f(n) = \Omega(n^{log_b(a)+\varepsilon})$ for some $\varepsilon > 0$
   and if $af(\frac{n}{b}) \leq cf(n)$ for $c < 1$ and sufficiently large n
   then you may conclude that $T(n) = \Theta(f(n))$

7.0.1 Master Theorem Examples: Case 1 illustration

$$T(n) = 9T(\frac{n}{3}) + n \rightarrow \quad a = 9 \quad b = 3 \quad f(n) = n$$

$$n^{log_b(a)} = n^{log_3(9)} = n^2 \rightarrow f(n) = O(n^{2-\varepsilon}) \quad \forall \quad 0 < \varepsilon \leq 1$$

Therefore this example falls into case 1 and

$$T(n) = \Theta(n^{log_b(a)}) = \Theta(n^{log_3(9)}) = \Theta(n^2)$$

7.0.2 Case 2 illustration

$$T(n) = 9T(\frac{2n}{3}) + 1 \rightarrow \quad a = 1 \quad b = \frac{3}{2} \quad f(n) = 1$$

$$n^{log_b(a)} = n^{log_{\frac{3}{2}}(1)} = n^0 \because any \quad log_x(1) = 0 \rightarrow f(n) = \Theta(n^0) = \Theta(1)$$

Therefore this example falls into case 2 and

$$T(n) = \Theta(n^0 log(n)) = \Theta(log(n))$$

7.0.3 Case 3 illustration

$$T(n) = 3T(\frac{n}{4}) + nlog(n) \rightarrow \quad a = 3 \quad b = 4 \quad f(n) = nlog(n)$$

$$n^{log_b(a)} = n^{log_4(3)} = n^0.7925 \rightarrow f(n) = \Omega(n^{.7925-\varepsilon}) \quad for \quad \varepsilon = 0.2$$

$$and \quad af(\frac{n}{b}) = 3f(\frac{n}{4}) = 3\frac{n}{4}log(\frac{n}{4}) \quad \therefore af(\frac{n}{b}) \leq cf(n)$$

Therefore this example falls into case 3 and

$$T(n) = \Theta(nlog(n))$$

8 Divide and Conquer

1. Divide: Find a way to separate sub-problems out of the original. Ideally, they should be smaller instances of the same fundamental problem.
2. Conquer: If the size of the problem is small enough solve it in a straightforward way, or, if the problem is not small enough, solve the sub-problem recursively.
3. Combine: Put together the solutions of the sub-problems in a way that yields the solution to the main problem

8.1 Merge Sort: The Recursive sorting algorithm

1. Divide: Break up the sequence of n elements into two smaller sub-sequences of length $\frac{n}{2}$
2. Conquer: If the sub-sequences have length one, they are sorted, if they have a length greater than one, recursively sort the sub-sequences.
3. Combine: Bring together the sorted sub-sequences in a manner that sorts them to obtain the original array, now sorted.

```
1  MergeSort(A, start, end)begin
2      if start<end then
3          m=⌊(start+end)/2⌋
4          MergeSort(A, start, m)
5          MergeSort(A, m+1, end)
6          Merge(A, start, m, end)
7      end
8  end
```

Algorithm 3: Merge Sort

```
1  Merge(A, start, m, end)begin
2      //Linear Compleity i.e. Θ(n)
3      i=start
4      j=m+1
5      k=1
6      create an array B of length end-start+1
7      while i≤m and j≤end do
8          if A[i]< then
9              B[k]=A[i]
10             k++
11             i++
12         end
13         else
14             B[k]=A[j]
15             k++
16             j++
17         end
18     end
19     while i≤m do
20         B[k]=A[i]
21         i++
22         k++
23     end
24     while j≤end do
25         B[k]=A[j]
26         j++
27         k++
28     end
29     Copy B in A
30 end
```

Algorithm 4: Merge

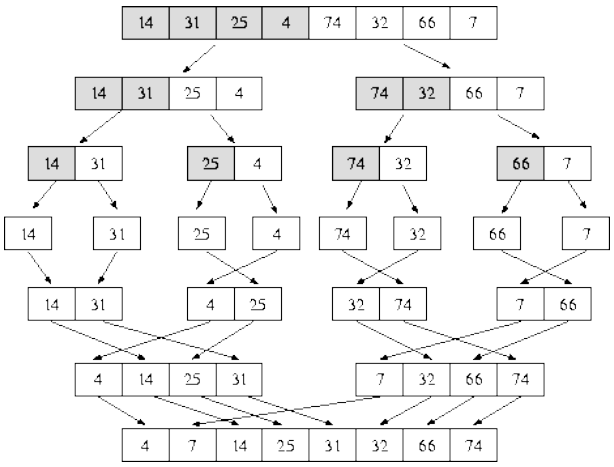


Figure 11: Merge Sort Execution

8.2 Merge Sort Complexity

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

Recursion Tree Method:

Tree	Level	Input Size	Cost
cn	0	n	cn
$\frac{cn}{2}$ $\frac{cn}{2}$	1	$\frac{n}{2}$	cn
$\frac{cn}{4}$ $\frac{cn}{4}$ $\frac{cn}{4}$ $\frac{cn}{4}$	2	$\frac{n}{4}$	cn
$\frac{cn}{2^i}$ $\cdots$ $\cdots$ $\frac{cn}{2^i}$	i	$\frac{n}{2^i}$	cn
$\Theta(1)$ $\cdots$ $\cdots$ $\Theta(1)$	k	$1 = \frac{n}{2^k}$ $k = log_2(n)$	2^k

Figure 12: Merge Sort Complexity Analysis: Recursion Tree Method

$$\sum_{i=0}^{log(n)-1} cn + c2^k = cn \sum_{i=0}^{log(n)-1} 1 + c2^{log_2(n)} = cnlog(n) + cn^{log_2(2)} = \Theta(nlog(n))$$

Master Theorem: a=2 b=2 f(n)=$\Theta(n)$

$$n^{log_b(a)} = n^{log_2(2)} = n^1 = n$$

$$case \quad 2 \because f(n) = \Theta(n^{log_2(2)}) = \Theta(n) = f(n)$$

$$\therefore T(n) = \Theta(nlog(n))$$

8.3 Another Complexity Example

$$T(n) = \begin{cases} 16T(\frac{n}{4}) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

Recursion Tree Method

Tree	Level	Input Size	Cost
cn	0	n	cn
$\frac{cn}{4} \cdots (16) \cdots \frac{cn}{4}$	1	$\frac{n}{4}$	$4cn$
$\frac{cn}{16} \cdots (16^2) \cdots \frac{cn}{16}$	2	$\frac{n}{16}$	$16cn$
$\frac{cn}{4^i} \cdots (16^i) \cdots \frac{cn}{4^i}$	i	$\frac{n}{2^i}$	$4^i cn$
$\Theta(1) \cdots (16^k) \cdots \Theta(1)$	k	$1 = \frac{n}{4^k} \quad k = log_4(n)$	$c16^k = c16^{log_4(n)} = cn^{log_4(16)} = cn^2$

Figure 13: Complexity Analysis: Recursion Tree Method

$$\sum_{i=0}^{log(n)-1} 4^i cn + c16^k = cn \sum_{i=0}^{log(n)-1} 4^i + cn^2$$

To solve summation recall:

$$\sum_{i=0}^{k} z^i = \frac{z^{k+1} - 1}{z - 1}$$

Implemented here:

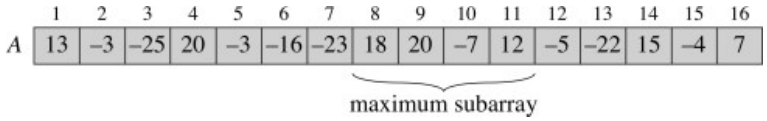
$$cn \sum_{i=0}^{log(n)-1} 4^i + cn^2 = cn\frac{4^{log_4(n)-1+1} - 1}{4 - 1}$$

Absorb denominator into general constant

$$cn \cdot n^{log_4(4)=1} - cn + cn^2 = cn^2 + cn^2 - cn = \Theta(n^2)$$

8.4 Maximum Subarray Problem

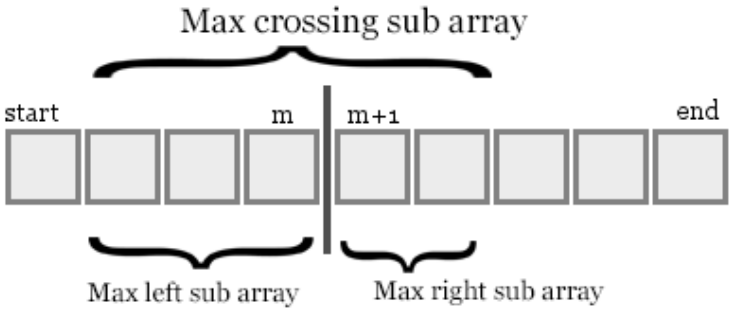
Given an array of size n containing positive and negative numbers find the subarray with the maximum sum of elements:



- Really dumb solution: Consider the first element and all possible subarrays beginning with that element summing each of these along the way until it proceeds to the next element and repeats all the calculations = $\Theta(n^3)$
- Straightforward (and inefficient) solution: Try every subarray and find the maximum = $\Theta(n^2)$
- Divide and Conquer Algorithm: allows the problem to be solved in $\Theta(nlog(n))$

Divide and Conquer approach

- Let i and j be the beginning and end of a subarray with max sum
- If the max subarray is on the left side, both i and j are on the left, $start \leq i \leq j \leq m \leq end$
- If the max subarray is on the right side, both i and j are on the right, $start \leq m+1 \leq i \leq j \leq end$
- If the max subarray crosses m it has elements on the left and right of m, $start \leq i \leq m \leq j \leq end$



FindMaxSubarray Complexity by Master Theorem

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

$$a = 2 \quad b = 2 \quad f(n) = n$$

$$n^{log_2(2)} = n \quad f(n) = n = \Theta(n) \quad \therefore case \quad 2$$

$$T(n) = \Theta(nlog(n))$$

8.5 QuickSort

Quicksort applies the divide and conquer approach to the sorting problem. The internal constants are known and proven to be small which implies that Quicksort is one of the best solutions in practical situations. In each successive recursive call one element is placed in the proper position and is called a pivot.

- Divide: given the array A[start..end] choose a value for the pivot x such that A[p]=x alter the array so both of the following statements hold:

$$\forall i \in [start...p-1] \quad A[i] \leq x$$
$$\forall i \in [p+1...end] \quad A[i] > x$$

- Conquer: Recursively sort the left and right sides of x. The subarrays A[start...p-1] and A[p+1...end]
- Combine: Nothing left to do. The array is already sorted.

```
1  FindMaxCrossingSubarray(A, start, m, end)begin
2      // Θ(n)
3    lsum=-∞ //sum of max left subarray ending with m
4    maxl //index of element that begins the max subarray
5    sum=0
6    for  i=m down to start do
7      sum = sum+A[i]
8      if lsum¡sum then
9        lsum=sum
10       maxl=i
11     end
12   end
13   rsum=-∞
14   maxr
15   sum=0
16   for  j=m+1 end do
17     sum = sum+A[j]
18     if rsum¡sum then
19       rsum=sum
20       maxr=i
21     end
22   end
23   return (maxl, maxr, lsum+rsum)
24 end
```

Algorithm 5: FindMaxCrossingSubarray

Complexity of QuickSort Assume an array of length n

$$T(n) = \begin{cases} T(k) + T(n-k-1) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

Worst case: x is the min $\rightarrow$ k=0 or x is the max $\rightarrow$ k=n-1 then Quicksort=$\Theta(n^2)$:

$$T(n) = \begin{cases} \cancel{T(0)} + T(n-1) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

```
1  FindMaxSubarray(A, start, end){
2  if start==end then
3  |  return (start, end, A[start])
4  end
5  m=⌊(start+end)/2⌋
6  (lstart, lend, lsum)=FindMaxSubarray(A, start, m)
7  (rstart, rend, rsum)=FindMaxSubarray(A, m+1, end)
8  (cstart, cend, csum) = FindMaxCrossingSubarray(A, start, m, end)
9  if lsum≥rsum and lsum≥csum then
10 |  return(lstart, lend, lsum
11 end
12 if rsum≥lsum and rsum≥csum then
13 |  return(rstart, rend, rsum
14 end
15 return(cstart, cend, csum)
16 }
```

Algorithm 6: FindMaxSubarray

Best case: k is the index of the median array element i.e. k$\approx \frac{n}{2}$ then QuickSort=$\Theta(nlog(n))$:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

```
1  QuickSort(A, start, end)begin
2  |  if start<end then
3  |  |  p=Partition(A, start, end)
4  |  |  QuickSort(A, start, p-1)
5  |  |  QuickSort(A, p+1, end)
6  |  end
7  end
```

Algorithm 7: QuickSort

```
1  Partition(A, start, end)begin
2  |  x=A[end]
3  |  i=start-1
4  |  for j=start to end do
5  |  |  if A[j]≤x then
6  |  |  |  i++
7  |  |  |  swap(A[i], A[j])
8  |  |  end
9  |  end
10 |  swap(A[i+1], A[end])
11 |  return i+1
12 end
```

Algorithm 8: Partition

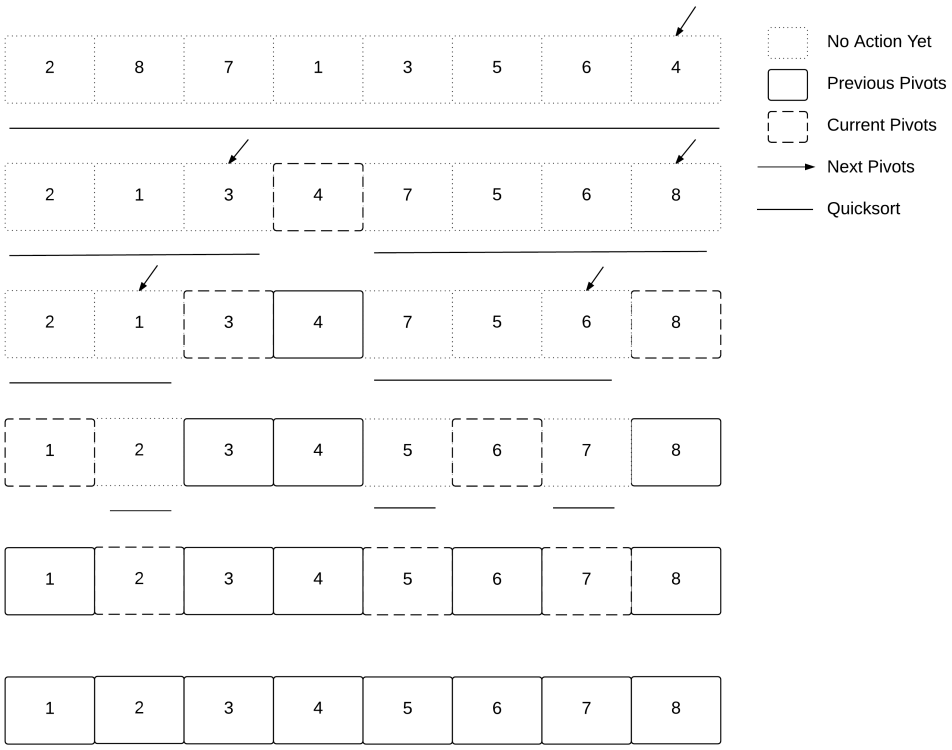


Figure 14: Execution of QuickSort on an array

8.6 Counting Sort

Counting Sort achieves sorting in linear time. The basic idea is, based on the assumption that the array to be sorted is of size n, and its maximum integer is k, CountingSort counts the number of distinct elements and this count determines the position of each element in the output array.

1. Create new array
2. Fill out the new counting array with the number of occurrences of each element
3. Update the counting array cumulatively, adding up occurrences

In the context of the following pseudocode, A is the input array, B is the output array, n is the number of elements in the array, and k is the maximum value of any individual array element.

```
1  CountingSort(A, B, n, k)begin
2  |  Let C[0...k] be a new array
3  |  for i=0 to k do
4  |  |  C[i]=0
5  |  end
6  |  for j=1 to n do
7  |  |  C[A[j]]=C[A[j]]+1
8  |  end
9  |  for i=1 to k do
10 |  |  C[i]=C[i]+C[i-1]
11 |  end
12 |  for j=n to 1 do
13 |  |  B[C[A[j]]]=A[j] C[A[j]]=C[A[j]]-1
14 |  end
15 end
```

Algorithm 9: CountingSort

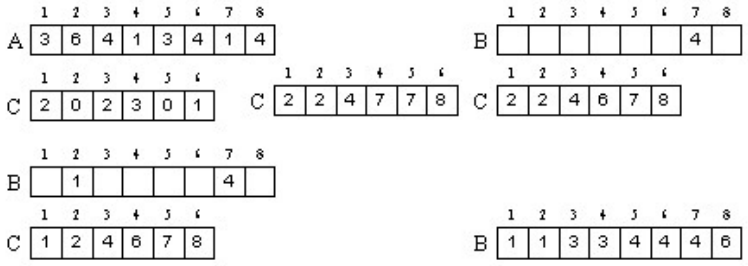


Figure 15: CountingSort Execution

Part II
Iterative Problem Solving

This strategy relies on the decomposition of the main problem into subproblems, but in a different manner than in the Divide and Conquer approach. This is particularly advantageous when subproblems are overlapped.

8.7 Examples

Fibonacci In the Fibonacci sequence, each successive value is based on both of the two preceding values, and nothing else.

$$F(x) = \begin{cases} 0 & x = 0 \\ 1 & x = 1 \\ F(n-1) + F(n-1) & x \geq 2 \end{cases}$$

Matrix Multiplication Given a matrix A 100x1, a matrix B 1x50, and a matrix C 50x2, consider the following cases:

$$A(BC) \rightarrow (100x1)((1x50)(50x2))$$

This series of multiplications includes the first parenthetical resolution requires 100 multiplication operations and then a further 200 multiplication operations, for a total of 300 multiplication operations to perform this calculation.

$$(AB)C \rightarrow ((100x1)(1x50))(50x2)$$

This series of multiplications is very different. The first resolution here requires 5,000 multiplication operations, and the second 10,000, for a total of 15,000 multiplication operations necessary in total. In both cases, the end result result is a 100x2 matrix, but the cost in time is vastly different.

9 Greedy Algorithms

Greedy Algorithms are useful in a wide array of cases, especially dealing with optimization problems defined in a particular domain. These are commonly problems of maximizing or minimizing some system parameter or variable. They are generally iterative approaches, and at each iteration the algorithm makes the "best" currently available decision according to its selection criteria. Greedy Algorithms do not always find the optimal solution to a problem. There are a set of conditions which are both necessary and sufficient for a problem to be optimally solved by a greedy algorithm.
If a greedy algorithm does not solve a problem optimally, often it provides a bounded solution with respect to the optimal solution.
Greedy Algorithms build a partial solution which is extended at each iteration, and it never changes it's mind.

9.1 The Activity Selection Problem

- Consider n activities $\in$ A=$[a_1 \cdots a_n]$ from which a subset should be selected to occur in one classroom
- Each activity has a start and finish time
- The i^{th} activity has a start time s_i and a finish time f_i such that $0 \leq s_i < f_i < \infty$.
- When an activity is scheduled, it takes place in the open interval $[s_i, f_i)$
- Two activities $a_1 = [s_i, f_i)$ and $a_2 = [s_j, f_j)$ are compatible if they do not overlap ie: $s_i \geq f_j$ or $s_j \geq f_i$

Problem: Given the activities in A, select the maximum number of compatible activities. Consider the following diagram representing the set of activities A. In this example, one optimal, but not the only optimal, solution would consist of the activities $\{a_1, a_3, a_6, a_8\}$. In any solution, given this set

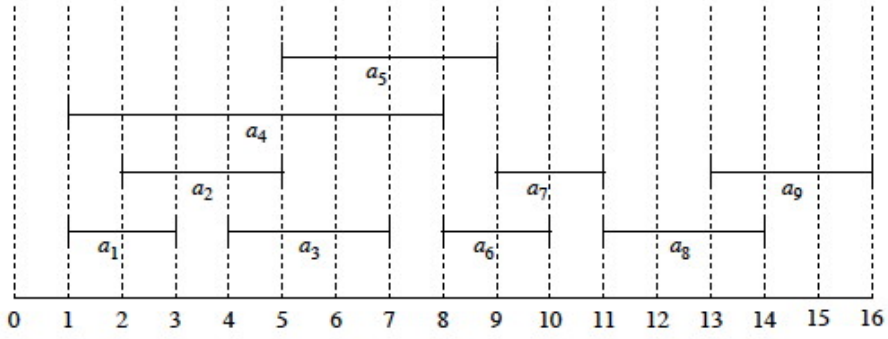


Figure 16: Example of a list of activities to which this Greedy algorithm may be applied

of activities, the maximal number which may be compatible is 4. Therefore any solution which consists of exactly 4 compatible activities is optimal. If instead the desired parameter to be optimized is amount of class time allocated, the problem becomes considerably more complex, and in fact cannot be solved in polynomial time.

The Greedy Approach - Exploring criteria: Iteratively build a set s which consists of compatible activities. The set s is extended at each iteration. In each iteration which activity should be selected? What should be the criteria by which the greedy algorithm makes its choice?

1. Earliest start time
2. Least duration
3. Earliest termination time
4. least conflicting activities

Counter Examples to Earliest Start Consider the following set of activities:

With the greedy criteria being earliest start, a_1 alone is chosen, since it conflicts with both of the other two potential activities. If one of the other activities had been chosen a max of two activities could have been selected.

```
1 EarliestStart(A)begin
2 |  S=∅ //the null or empty set
3 |  while ∃ a ∈ A : S ∪{a} does not create overlaps do
4 |  |  D=subset of (A-S) that do not overlap with activities in S
5 |  |  a_k = arg min_{a_i∈D} s_i
6 |  |  S=S∪{a_k}
7 |  end
8 |  return S
9 end
```

Algorithm 10: Earliest Start time as greedy criteria

a_1	[1,10)
a_2	[2,3)
a_3	[4,5)

Figure 17: Activity set to serve as counterexample to EarliestStart

This selection criteria therefore does not lead to an optimal solution, proven by counter example.

Counter Examples to Earliest Start Consider the following set of activities:

a_1	[5,7)
a_2	[3,6)
a_3	[6,9)

Figure 18: Activity set to serve as counterexample to LeastDuration

With the greedy criteria being least duration, a_1 alone is again chosen, since it conflicts with both of the other two potential activities. If one of the other activities had been chosen a max of two activities could have been selected. This selection criteria therefore does not lead to an optimal solution, proven by counter example.

```
1 LeastDuration(A)begin
2 |  S=∅ //the null or empty set
3 |  while ∃ a ∈ A : S ∪{a} does not create overlaps do
4 |  |  D=subset of (A-S) that do not overlap with activities in S
5 |  |  a_k = arg min_{a_i∈D} (f_i − s_i) //This is the only altered line
6 |  |  S=S∪{a_k}
7 |  end
8 |  return S
9 end
```

Algorithm 11: Least Duration as greedy criteria

```
1 EarliestTermination(A)begin
2 |  S=∅ //the null or empty set
3 |  while ∃ a ∈ A : S ∪{a} does not create overlaps do
4 |  |  D=subset of (A-S) that do not overlap with activities in S
5 |  |  a_k = arg min_{a_i∈D} f_i
6 |  |  S=S∪{a_k}
7 |  end
8 |  return S
9 end
```

Algorithm 12: Earliest Finish time as greedy criteria

Earlier Counter Examples Handled Consider the following set of activities:

For Set 1 and Set 2, with the other criteria, only one activity (a_1) could be selected, but here, since a_2 has the earliest termination time it is selected first, and in the next iteration, a_3 is selected which is the only other available, non-conflicting activity. It yields an optimal solution, in both cases.

9.2 Greedy Algorithm Correctness

The proof of correctness for a greedy algorithm is a multistage process, but three main steps are required.

	Set 1	Set 2
a_1	[1,10)	[5,7)
a_2	[2,3)	[3,6)
a_3	[4,5)	[6,9)

Figure 19: Activity set to serve as counterexample to EarliestStart

1. Termination - Prove that for any input, it will complete a finite number of steps
2. Partial Solution Correctness - Prove that the partial solution built at every iteration is included in the optimal solution. This is normally achieved through a proof by induction.
3. Demonstrate that the final solution is optimal

Application to Earliest Termination

1. We add a new activity in s at every iteration of the loop so we can have, at most, $|A|$ iterations.
2. Base: (h=0) 0^{th} iteration this implies that $S_0 = \emptyset = \{\}$. The empty set is a subset of any set so $S_0 \subset S^* \; \forall$ optimal solution S^*
   Assume: at iteration $h > 0$ $S_h \subset S^*$
   Inductive Step: There exists an optimal solution such that S_{h+1} is a subset of it.
   At iteration h+1, $S_{h+1} = S_h \cup a_k$
   If $a_k \in S^*$ then $S_{h+1} \subseteq S^* \rightarrow$ done
   If $a_k \notin S^*$ then $S_{h+1} \nsubseteq S^* \rightarrow$
   It must therefore be proven that there exists a different optimal solution $S^\#$ such that $S_{h+1} \subseteq S^\#$ Facts:
   - S_{h+1} does not contain overlaps
   - $|S^*| \geq |S_{h+1}| \rightarrow \exists a_j \in S^* \wedge \notin S_{h+1}$
   - Let $a_j \in \{S^* \setminus S_{h+1}\}$ be the activity with the earliest termination time
   - $a_j \in S_h$ also $S_h \subseteq S^*$ and $a_j \in S^\#$ and S^* does not contain overlaps

- Therefore a_j does not create overlaps with activities in S_h and would have been selected, but the algorithm selected $a_k \therefore f_k \leq f_j$
- $S^{\#} = \{S^* \setminus \{a_j\} \cup \{a_k\}\}$
- this does not contain overlapp becuse a_k has the same or earlier finish time than a_j
- Therefore $S^{\#}$ is optimal if S^* is because $|S^{\#}| = |S^*|$

3. Demonstrate final Solution is optimal:
   Let m be the number of iterations of the while loop.
   Proof by contradiction:

$$\exists S^* : S_m \subseteq S^*$$

   Assume falsity of the prior statement

$$|S_m| < |S^*|$$

$$\exists a : a \in S^* \wedge a \notin S_m$$

   Therefore a is compatible with every activity in S^* and if a $\in S^*$ then it also does not overlap with any activities in $S_m \because S_m \subseteq S^*$
   Therefore the while loop should execute once more to include a
   This is a contradiction. Therefore S_m is optimal and $|S_m| = |S^*|$

9.3 The Cashier Problem

A Cashier has to give W cents (integer) in change for a transaction. Optimize the return of coinage for the least number of coins being used in the make up of the proper amount of change. The available coin types/values belong to the set $C = \{c_1 \dots c_n\}$ and every element of C has an associated value V such that $c_i = v_i$. Formally:

$$S^* = \arg\min_{S \subseteq C}(|S|) \quad repeated : \sum_{c_i \in S} v_i = W$$

Examples in the US System C=1, 5, 10, 25

1. W=30 $\rightarrow S^* = \{25, 5\}$
2. W=67 $\rightarrow S^* = \{25, 25, 10, 5, 1, 1\}$

Greedy Solution: Pick the largest value coin with value less than or equal to W, then update remaining value of W.

```
1 Change(W)begin
2 |  s=∅
3 |  while W > 0 do
4 |  |  c_k = arg max_{c_i∈C:v_i≤W} (v_i)
5 |  |  s = s ∪ {c_k}
6 |  |  W = W − v_k
7 |  end
8 |  return s
9 end
```

Algorithm 13: Change Algorithm

Optimality This algorithm will yield an optimal solution for the US system of coinage, but it is not guaranteed to do so for all systems. Consider the US coinage system without the nickel, and making change of the amount 30 cents. The yielded solution is not optimal. The execution of the algorithm returns an set as follows: {25, 1, 1, 1, 1, 1} with 5 elements when the optimal solution would have been this set: {10, 10, 10} which has 3 elements.

9.4 The Knapsack Problem

A thief has a sack which can hold a limited amount of weight. In a shop, at night, he must select to steal those items which his sack can hold and maximize his 'profit'.

- A set $A = \{a_1, a_2 \dots a_n\}$ items.
- For all a_i there exists both a value v_i and a weight w_i.
- The knapsack has a capacity related to the amount of weight it can contain W.
- The thief has to find the set of items (S^*) that maximizes the overall value of items that can, concurrently, be contained in the knapsack.

$$S^* = \arg\max_{S \subseteq A}(\sum_{a_i \in S} w_i :\leq W)$$

The Greedy approach to the Knapsack: Start with an empty knapsack, $S = \emptyset$, and at each iteration add the item which meets one of the following choices for a greedy criteria:

- max v_i
- min w_i
- max $\frac{v_i}{w_i}$

```
1  GreedyKnapsack(A)begin
2  |  S=∅ //the null or empty set
3  |  R=W //R is the residual weight capacity of the knapsack
4  |  D=A while D ≠ 0 do
5  |  |  a_k = arg max_{a_i∈D} v_i | arg max_{a_i∈D} v_i/w_i | arg min_{a_i∈D} w_i
6  |  |  if w_k ≤ R then
7  |  |  |  S=S∪{a_k}
8  |  |  |  R = R − w_k
9  |  |  end
10 |  |  D = D − a_k
11 |  end
12 |  return S
13 end
```

Algorithm 14: Greedy Knapsack

Counter Examples for each criteria:

1. max v_i
   Consider the set of three items which consists of one item with weight equal to the capacity of the knapsack and value 100 as well as two additional items each of weight $\frac{W}{2}$ and value 99. The optimal solution would be to take the two 99 valued items since their collective weight

fits in the knapsack and the value is 198 rather than the single item with value 100. It would be the single more valuable item that the algorithm would select however, with this criteria.

2. min w_i
   Consider the set of three items which consists of one item with weight equal to the capacity of the knapsack and value 100 as well as two additional items each of weight $\frac{W}{2}$ and value 30. The optimal solution would be to take the single item with value 100 rather than the two 30 valued items, a total value of only 60, since it's collective weight fits in the knapsack and the value is 100. It would be the two less hefty items that the algorithm would select however, with this criteria.

3. max $\frac{v_i}{w_i}$
   Consider the set of three items which consists of one item with weight 60 and value 120 as well as two additional items each with weight 50 and value 99. Assuming the capacity of the knapsack is 100, the optimal solution would be to take the two 99 valued items since their collective weight fits in the knapsack and the value is 198 rather than the single item with value 120. It would be the single item with the $\frac{v_i}{w_i}$ ratio of 2 that the algorithm would select however, with this criteria, rather than the other two since their ratio is less than 2.

Variations on the Knapsack Problem

- Unbounded Knapsack: There are infinite quantities of each item available
- Bounded Knapsack: Finite quantities of each item, but multiples are permitted.

In all of these situations, the greedy approach does not work. It can and in some cases will yield a solution which is at least half of the optimal solution. In the knapsack problem this occurs only for the ratio approach to the selection criteria.

Part III
Dynamic Programming

Dynamic programming solves optimization problems by combining solutions of subproblems, but the subproblems are not disjoint, therefore individual subproblems may be common components to multiple higher level subproblems. To avoid computing the solution to the same subproblem multiple times, a table is maintained with the solutions already calculated and the final solution is acquired by a bottom up reading of the table, once it has been entirely populated with meaningful subproblem solutions.

10 General Dynamic Programming Approach

When an algorithm is designed using a dynamic programming approach, three steps are involved:

1. Find subproblems such that their solution can lead to the solution of the bigger problem.
    - The number of subproblems should be polynomial with respect to the original problem.
    - Polynomial complexity is also a requisite in the process to combine the solutions to obtain the solution to the original problem.
2. Define the recurrence relation for the solution of the subproblems and the overall problem.
3. First focus on calculating the value of the solution then on calculating the actual solution.

11 Dynamic Programming Algorithms

11.1 The Dynamic Max Sub Array Problem

Given an array A with positive and negative numbers find the subarray with maximum sum. Following are the complexities of various types of approaches:

- Brute Force - $\Theta(n^3)$
- Trivial - $\Theta(n^2)$
- Divide and Conquer - $\Theta(nlogn)$
- Dynamic Programming - $\Theta(n)$

Definition 11.1 (The table T involved in Dynamic Programming Algorithms)

The table T (in this case an array) is the table in which partial solutions are stored
T[i] is the sum of the elements in the max subarray that ends at i with i=1..n
The maximum value in the array T is the solution.

Recursive relation to calculate T[i]:

$$T[i] = \begin{cases} A[1] & i = 1 \\ max(A[i], A[i] + T[i-1]) & i > 1 \end{cases}$$

11.2 Dynamic Programming Execution Example

Find the Maximum Subarray within $A =< -1, 2, 10, -13, 5, -10, 1, -2, -4 >$

A	-1	2	10	-13	5	-10	1	-2	-4
T	-1	2	12	-1	5	-5	1	-1	-4
max	-1	2	12	12	12	12	12	12	12
i	1	2	3	4	5	6	7	8	9

Figure 20: Step by step execution on the given array

11.3 MaxSubarray Pseudo Code

```
1  MaxSubarrayDP(A) begin
2  |  T[1]=A[1]
3  |  max = T[1]
4  |  b=1
5  |  for 1=2 to n do
6  |  |  if T[i − 1] < 0 then
7  |  |  |  T[i]=A[i]
8  |  |  end
9  |  |  else
10 |  |  |  T[i]=A[i]+T[i-1]
11 |  |  end
12 |  |  if max < T[i] then
13 |  |  |  max = T[i]
14 |  |  |  b=i
15 |  |  end
16 |  end
17 |  return max
18 end
```

Algorithm 15: Dynamic Programming MaxSubarray Algorithm

In the context of the preceding algorithm, b is defined as the last element of the max subarray. Every time max is updated, b also must be updated to find the max subarray's beginning index, start at b and sum in the array. Following is a different way to handle the return information.

```
1  MaxSubarrayDP(A) {
2  T[1]=A[1]
3  max = T[1]
4  b=1
5  for 1=2 to n do
6  |  if T[i − 1] < 0 then
7  |  |  T[i]=A[i]
8  |  end
9  |  else
10 |  |  T[i]=A[i]+T[i-1]
11 |  end
12 |  if max < T[i] then
13 |  |  max = T[i]
14 |  |  b=i
15 |  end
16 end
17 a=b
18 while max − A[a] ≠ 0 do
19 |  max=max-A[a]
20 |  a–
21 end
22 return (a, b)
23 }
```

Algorithm 16: Dynamic Programming MaxSubarray Algorithm with more useful return values

11.4 Longest Common Subsequence

Definition 11.2 (Subsequence)
Given a sequence $x =< x_1, x_2, \ldots x_n >$, $z =< z_1, z_2, \ldots z_k >$ *is a subsequence of* x *if there exists a strictly increasing list of indices* $< i_1 \ldots i_k >$ *such that for all* j $i_j \in [1 \ldots n]$ *and* $i_j < i_{j+1}$ *and for all* $j \in [i \ldots k]$, $x_j == z_j$

Subsequence Example $x =< 9, 15, 3, 6, 4, 2, 5, 10, 3 >$
$z =< 15, 6, 2 >$ z is a subsequence of x
$y =< 4, 3, 10 >$ y is not a subsequence of x

Example Problem: Given two sequences $x =< x_1 \dots x_n >$ and $y =< y_1 \dots y_m >$ find the longest common subsequence (LCS)

$$x =< 9, 15, 3, 6, 4, 2, 5, 10, 3 >$$
$$y =< 8, 15, 6, 7, 9, 2, 11, 3, 1 >$$
$$LCS =< 15, 6, 2, 3 >$$
$$x =< 9, \underline{15}, 3, \underline{6}, 4, \underline{2}, 5, 10, \underline{3} > \quad indices < 2, 4, 6, 9 >$$
$$y =< 8, \underline{15}, \underline{6}, 7, 9, \underline{2}, 11, \underline{3}, 1 > \quad indices < 2, 3, 6, 8 >$$

Definition 11.3 (Prefix x_i) *Given a sequence $x =< x_1 \dots x_n >$ its prefix x_i is defined as $< x_1, \dots x_i > \forall i \in [1 \dots n]$*

Definition 11.4 (Table T) *$\forall i \in [1 \dots n] \wedge j \in [1 \dots m]$, $T[i, j]$ is the length of the longest common subsequence of x_i and y_j*

In the context of this approach to the problem and the algorithm which will be defined, there exists two set of base cases: $T[i, 0] = 0$ and $T[0, j] = 0$
When both i and j are greater than zero there are two other cases that need to be defined.

1. if x[i] is equal to y[j] then these two elements are in common and will be a part of a common subsequence. The table element relating to these respective sequence elements must therefore be appropriately incremented i.e. T[i,j]=T[i-1,j-1]+1

2. if x[i] is not equal to y[j] then these elements do not belong to a common subsequence and T[i,j]=max(T[i-1,j],T[i,j-1])

Summarized formally the function defining the table T is as follows:

$$T[i, j] = \begin{cases} 0 & i = 0 \vee j = 0 \\ T[i-1, j-1] & x[i] = y[j] \\ max(T[i-1, j], T[i, j-1]) & x[i] \neq y[j] \end{cases}$$

```
1  LCS(X,Y) begin
2      for i=0 to n do
3          T[i,0]=0
4      end
5      for j=0 to m do
6          T[0,j]=0
7      end
8      for i=1 to n do
9          for j=1 to m do
10             if x[i]==y[j] then
11                 T[i,j]=T[i-1,j-1]+1
12             end
13             else
14                 T[i,j]=max(T[i-1,j],T[i,j-1])
15             end
16         end
17     end
18     return T[n,m]
19 end
```

Algorithm 17: Dynamic Programming LCS Algorithm

11.5 LCS Pseudo Code

LCS Execution Example with the table T Find the LCS in the two sequences following:

$$x =< 9, 15, 3, 6, 4, 2, 5, 10, 3 >$$

$$y =< 8, 15, 6, 7, 9, 2, 11, 3, 1 >$$

The longest common subsequence has a length of 4, but how can the LCS

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1
2	0	0	1	1	1	1	1	1	1	1
3	0	0	1	1	1	1	1	1	2	2
4	0	0	1	2	2	2	2	2	2	2
5	0	0	1	2	2	2	2	2	2	2
6	0	0	1	2	2	2	3	3	3	3
7	0	0	1	2	2	2	3	3	3	3
8	0	0	1	2	2	2	3	3	3	3
9	0	0	1	2	2	2	3	3	4	4

Figure 21: Example table T filled in execution of the LCS algorithm

be found explicitly from the table rather than only this indirect information regarding its length?

11.6 PrintLCS Pseudo Code and Execution

From the bottom right cell of the table T the PrintLCS algorithm works its way toward the top left cell along the elements that form the LCS. Follow the boldface, underlined or italicized elements in the following table to trace the LCS through the table. Elements that have been put in boldface are members of the trace back through the table and are given to show completeness of the execution of the PrintLCS function. Elements which are underlined mark indices of elements which will be output or printed by the algorithm. The italicized element indicates the point at which the execution of the PrintLCS function will terminate:

```
1  PrintLCS(T,i,j)begin
2      if i¿0 and j¿0 then
3          if x[i]==y[j] then
4              PrintLCS(T, i-1,j-1)
5              Print(x[i])
6          end
7          else
8              if T[i − 1, j] ≥ T[i, j − 1] then
9                  PrintLCS(T,i-1,j]
10             end
11             else
12                 PrintLCS(T,i,j-1)
13             end
14         end
15     end
16 end
```

Algorithm 18: Printing the elements of the LCS

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	***0***	0	0	0	1	1	1	1	1
2	0	0	**1**	1	1	1	1	1	1	1
3	0	0	**1**	1	1	1	1	1	2	2
4	0	0	1	**2**	**2**	**2**	**2**	2	2	2
5	0	0	1	2	2	2	**2**	2	2	2
6	0	0	1	2	2	2	3	**3**	3	3
7	0	0	1	2	2	2	3	**3**	3	3
8	0	0	1	2	2	2	3	**3**	3	3
9	0	0	1	2	2	2	3	3	**4**	**4**

Figure 22: Traceback through T of the LCS in the PrintLCS algorithm

11.7 Dynamic Programming Knapsack

Problem Description: There exists a set of items A, $\{a_1 \dots a_n\}$ with values $\{v_1 \dots v_n\}$ and weights $\{w_1 \dots w_n\}$. Given a knapsack of capacity W, find a set of items S with the maximum possible value which can fit in the knapsack.

We define a table T of dimensions $n \times w$ such that T[i,j] is the value of the solution of the knapsack problem considering the first i items in A and a knapsack of capacity j. Recursively assume that the problem has been solved for all indices less than i and j respectively. When filling the table element T[i,j] at the i^{th} step element a_i is considered. If w_i is greater than j then T[i,j] = T[i-1,j]. If a_i is in the solution then $T[i,j] = v_i + T[i-1, j-w_i]$ and if a_i is not in the solution, T[i,j]=T[i-1,j]. Formally, $\forall j \in \{1 \dots w\} \wedge i \in \{1 \dots n\}$:

$$T[i,j] = \begin{cases} 0 & i = 0 \vee j = 0 \\ T[i-1,l] & w_i > j \\ max(v_i + T[i-1, j-w_i], T[i-1,j]) & i > 0 \wedge j > 0 \wedge w_i \leq j \end{cases}$$

11.8 KnapsackDP Pseudo Code

```
1  KnapsackDP(A, W)begin
2      for i=1 to n do
3          T[i,0]=0
4      end
5      for j=i to W do
6          T[0,j]=0
7      end
8      for i=1 to n do
9          for j=1 to W do
10             if w_i > j then
11                 T[i,j]=T[i-1,j]
12             end
13             else
14                 T[i,j]=max(T[i-1,j], v_i + T[i-1,j-w_i])
15             end
16         end
17     end
18     return T[n,W]
19 end
```

Algorithm 19: KnapsackDP Pseudo Code

11.9 KnapsackDP Example

Execute the KnapsackDP algorithm on the following set of items with their associated values given here:

Item	Value	Weight
a_1	1	1
a_2	6	2
a_3	18	5
a_4	22	6
a_5	28	7

Figure 23: Set of Items A upon which to execute the KnapsackDP algorithm

$A \setminus W$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	40

Figure 24: The table T after the execution of the algorithm on the given set of items

11.10 Backtrack Knapsack Pseudo Code

Note that initially i=n and j=W.

```
1  PrintKnapsack(A, T, i, j)begin
2      if i ≤ 0 ∨ j ≤ 0 then
3          return
4      end
5      if T[i, j] == v_i + T[i − 1, j − w_i] then
6          print a_i
7          PrintKnapsack(A, T, i − 1, j − w_i)
8      end
9      else
10         PrintKnapsack(A, T, i − 1, j)
11     end
12 end
```

Algorithm 20: PrintKnapsack Pseudo Code

In a similar manner to the previous backtracking exploits, elements that have been put in boldface are members of the trace back through the table and are given to show completeness of the execution of the PrintKnapsack function. Elements which are underlined mark indices of elements which will be output or printed by the algorithm. The italicized element indicates the point at which the execution of the PrintKnapsack function will terminate:

$A \setminus W$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	***0***	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	**18**	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	**40**
5	0	1	6	7	7	18	22	28	29	34	35	**40**

Figure 25: Traceback through T of KnapsackDP in the PrintKnapsack algorithm

Part IV
Graphs

12 Definitions

Definition 12.1 (Graph)
A graph is a pair of sets G=(V,E) in which V is the set of nodes or vertices and E the set of edges, also called links, such that $E \subseteq V \times V$

Definition 12.2 (Undirected Graphs)
A graph in which edges are bidirectional.

Definition 12.3 (Directed Graphs)
A graph in which edges are unidirectional.

Definition 12.4 (Degree: $deg(v)$**)**
Given an undirected graph G=(V,E) the degree of a node v in V is defined as the number of edges incident in v.
$deg(v) = |\{\{v, u\} : u \in V \wedge ((v, u) \in E \vee (u, v) \in E)\}|$

Definition 12.5 (Degree in Directed Graphs $deg^-(v)$ **and** $deg^+(v)$**)**
$deg^-(v)$ *is referred to as the in degree and is the number of incoming edges to a node.* $deg^-(v) = |\{\{v, u\} : u \in V \wedge (u, v) \in E\}|$
$deg^+(v)$ *is referred to as the out degree and is the number of outgoing edges from a node.* $deg(v) = |\{\{v, u\} : u \in V \wedge (v, u) \in E\}|$

Definition 12.6 (Walk)
Given a graph G=(V,E) a walk is a list of vertices $(v_1 \dots v_k)$ *such that* $(v_i, v_{i+1}) \in E, \forall i \in \mathbb{Z}_{k-1}$

Definition 12.7 (Path)
Given a graph G=(V,E) a path p is a walk such that each node except the first and last must not be a repeated use of another node, or, formally, the set of vertices must satisfy the following:

$$v_i \neq v_j if (i \neq j \wedge (j, i \neq k \vee j, i \neq 1))$$

Definition 12.8 (Cycle)
A cycle is a path such that $v_1 = v_k$.

Definition 12.9 (Connected Nodes)
Two nodes are connected if there exists a path $(v_1 \dots v_k)$ *such that* $u = v_1$ *and* $v = v_k$.

Definition 12.10 (Connected Component)
The component C *is a set of nodes such that* $C \subseteq V : \forall (u, v) \in C$ *they are connected.*

Definition 12.11 (Connected Graph)
A graph is connected if $\forall u, v \in V$ *they are connected.*

Definition 12.12 (Distance)
Distance is defined between two nodes as the length of the shortest path which exists between them. It is therefore a piece-wise function with two cases:

1. *If there exists a path between u and v,*
   (a) *If the graph is weighted d(u,v)= the sum of the weights between all the edges in the minimal path between u and v.*
   (b) *Else d(u,v)= the minimal number of edges between u and v*
2. *If there exists no path between u and v, d(u,v)=*∞

Definition 12.13 (Tree)
A graph is a tree if it is both connected, and contains no cycles

Definition 12.14 (Complete (Clique))
A graph is complete, or is a clique, if it contains all possible edges between its nodes.

13 Representation of Graphs

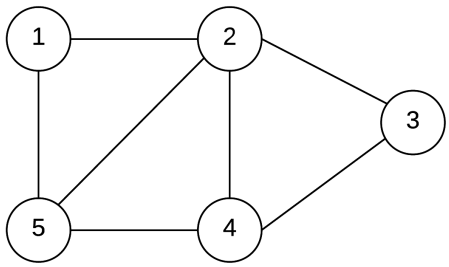


Figure 26: Example undirected graph

13.1 Adjacency Lists

There exists an array of size $|V|$, and each element in the array points to a list of adjacent elements in the graph to that node.

$$Adj[v] = \text{list of all vertices u} : (v, u) \in E$$

If the graph is directed, there will be $|E|$ elements in the list. If the graph is

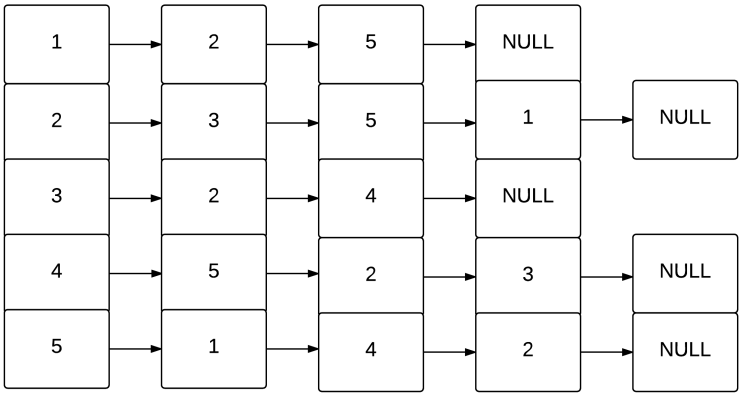


Figure 27: Example of an Adjacency List for the above graph

undirected, there will be $2|E|$ elements in the list. The memory complexity is $\Theta(|V| + |E|)$.

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Figure 28: Example of an Adjacency Matrix for the above graph

13.2 Adjacency Matrix

Matrix $M = |V| \times |V|$

$$M[i,j] = \begin{cases} 1 & (i,j) \in E \\ 0 & else \end{cases}$$

If the graph is undirected, the resulting associated matrix is symmetric about the diagonal, which means for all i and j, M[i,j]=M[j,i]. If the graph is directed, this no longer holds. This method has a memory complexity $\Theta(|V|^2)$

13.3 Representation comparison

	Adjacency List	Adjacency Matrix
Degree of a node	$\Theta(deg(v))$	$\Theta(\|V\|)$
$\exists(u,v) \in E$	$\Theta(deg(v))$	$\Theta(1)$

Figure 29: Basic Operations Cost Table

14 Depth First Search - DFS

Visit a graph

- The visit goes deeper and deeper in the graph until it can't find any non-visited nodes then it backtracks to visit other nodes.
- Keep track of the predecessor of each visited node.

- If the graph G_π, induced by the set of predecessors, is considered, then G_π is a forest, and, specifically, a DFS forest.
- Each node will consist of 2 attributes:
  - node.π - the predecessor of the node in question
  - node.visited (or node.v) - a boolean variable which is false until the node is visited in the course of algorithm execution.
- The complexity is $\Theta(|V| + \sum_{u \in V} deg(u))$ or $\Theta(|V| + |E|)$

14.1 DFS Pseudo code

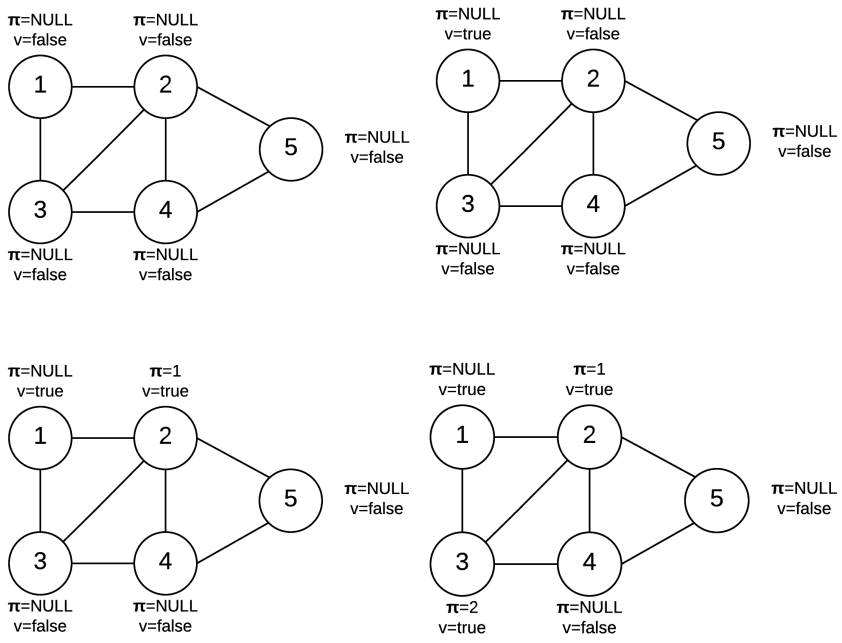
```
1  DFS(G) begin
2  |   for u ∈ V do
3  |   |   u.π =NULL
4  |   |   u.v =false
5  |   end
6  |   for u ∈ V : u.v == false do
7  |   |   //for a connected graph this is called once
8  |   |   DFSVisit(G,u)
9  |   end
10 end
```

Algorithm 21: Depth First Search Pseudo Code

```
1 DFSVisit(G,u) begin
2 |   u.v=true
3 |   for v ∈ Adj(u) do
4 |   |   if v.v==false then
5 |   |   |   v.π=u
6 |   |   |   DFSVisit(G,v)
7 |   |   end
8 |   end
9 end
```

Algorithm 22: Depth First Search Visit Pseudo Code

14.2 DFS Execution Example



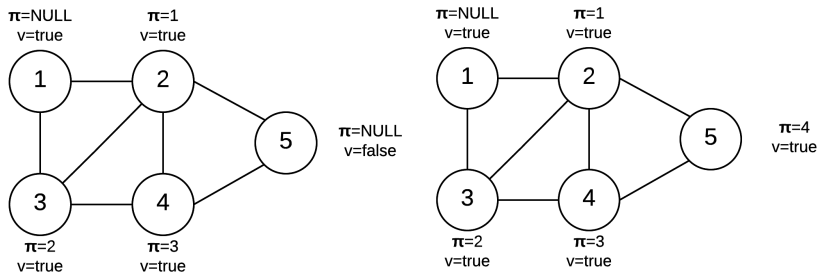


Figure 30: Step by step execution of the DFS Algorithm

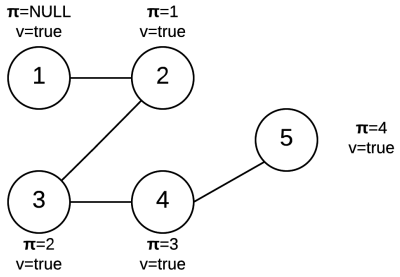


Figure 31: Graph pf the G_π Tree

15 Directed Acyclic Graphs (DAG)

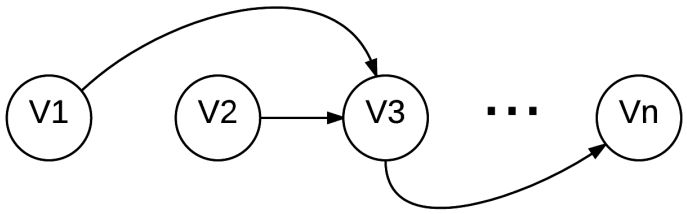
- Nodes are jobs or activities
- There exists an edge between u and v (u,v) if v cannot be executed until u has terminated
- because the graph has no cycles, there must exist at least one activity in degree 0
- Find a scheduling of activities $v'_1, v'_2, \ldots v'_n$ such that if the activities are executed in the prescribed order, when a general activity is reached, all the activities on which it depends have already been completed

Topological Sorting

- In the graph representing this, all edges go from left to right.
- Topological sorting modifies DFS to add two new attributes

- Start (s) - the time at which the visit of $u \in V$ begins.
- End (e) - the time at which the visit of DFS from u ends.

- Topological sort is obtained by sorting the nodes in descending order by end time




```
1  TopologicalDFS(G)begin
2  |   t=0
3  |   for u ∈ V do
4  |   |   u.π=NULL
5  |   |   u.v=false
6  |   |   u.start=0
7  |   |   u.end=0
8  |   end
9  |   for u ∈ V : u.v == false do
10 |   |   TDFSVisit(G,u)
11 |   end
12 end
```

Algorithm 23: TopologicalDFS Pseudo Code

```
1  TDFSVisit(G,u)begin
2  |   t=t+1
3  |   u.start=t
4  |   u.v=true
5  |   for v ∈ adj(u) do
6  |   |   if v.v==false then
7  |   |   |   v.π=u
8  |   |   |   TDFSVisit(G,v)
9  |   |   end
10 |   end
11 |   t=t+1
12 |   u.end=t
13 end
```

Algorithm 24: TDFSVisit Pseudo Code

```
1  TopologicalSort(G)begin
2  |   TopologicalDFS(G)
3  |   Sort nodes by decreasing end time
4  end
```

Algorithm 25: Overview of Topological Sorting Process Pseudo Code

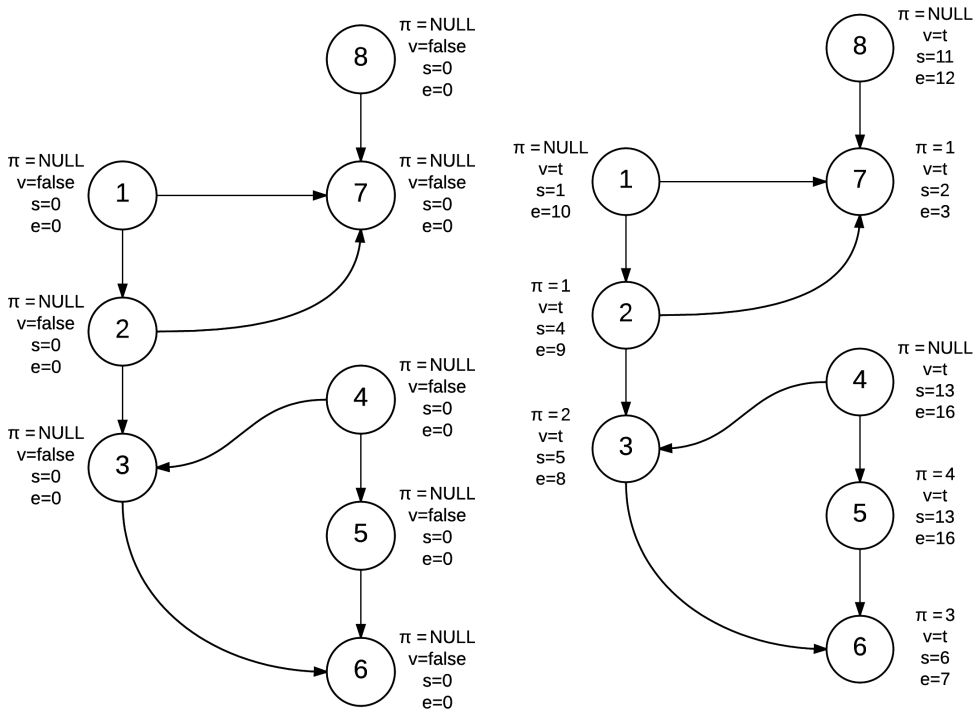


Figure 32: Initial and final states for the DAG on which the Topological Sorting algorithm is being executed

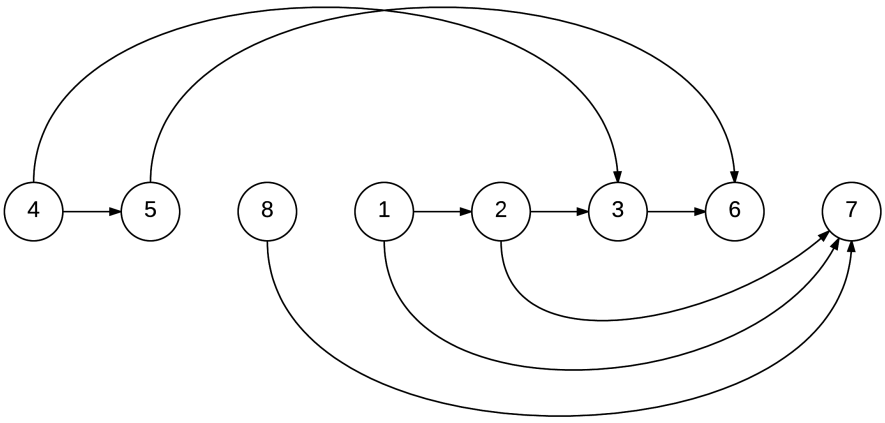


Figure 33: Final sorted order of the activities according to the Topological Sorting algorithm's execution

16 Breadth First Search (BFS)

Breadth first search is very similar to depth first search, with the exception that precedence is given to a nodes set of adjacent nodes before going as deep as possible in a graph. Also, one additional attribute is used:

- u.v, same as previously
- u.π, same as previously
- u.d - the distance from the source to the node in question

Complexity of the Breadth first search: Nodes may be enqueued only once, and the iteration of the internal while loop will happen, at most, $|V|$ times. Enqueue and Dequeue operations are assumed to be constant i.e. $\Theta(1)$, therefore:

$$BFS = \Theta(|V| + \sum_{u \in V} deg(u)) = \Theta(|V| + |E|)$$

16.1 BFS Pseudo Code

```
1  BFS(G,s)begin
2      for u ∈ V \ {s} do
3          u.v=false
4          u.d=∞
5          u.π=NULL
6      end
7      s.v=true
8      s.d=0
9      s.π=NULL
10     Q={}
11     Enqueue(Q,s)
12     while Q ≠ {} do
13         u=Dequeue(Q)
14         for v ∈ adj(u) do
15             if v.v==false then
16                 v.v=true
17                 v.d=u.d+1
18                 v.π=u
19                 Enqueue(Q,v)
20             end
21         end
22     end
23 end
```

Algorithm 26: Breadth First Search Pseudocode

16.2 BFS Example

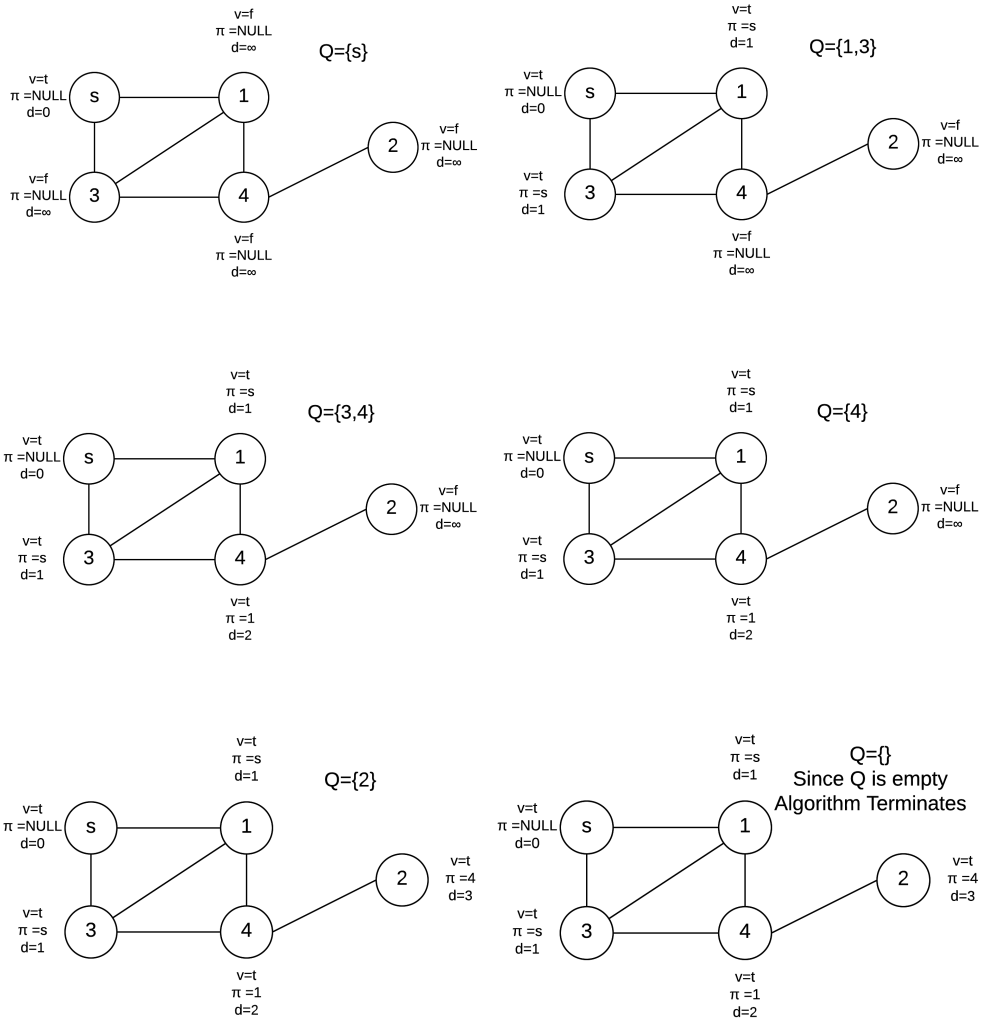


Figure 34: Step by step execution of the BFS Algorithm

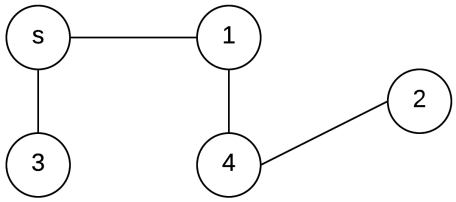


Figure 35: BFS Tree following execution of the algorithm

17 Strongly Connected Components

Definition 17.1 (Strongly Connected Graph)
Directed graphs are strongly connected if, for each pair of nodes, u and v, in V there exists a path from u to v and from v to u.

Definition 17.2 (Strongly Connected Component)
A strongly connected component c is a subset of V, which is also a subset of the graph, for which the previous definition's property holds, in a manner which describes a maximal subset.

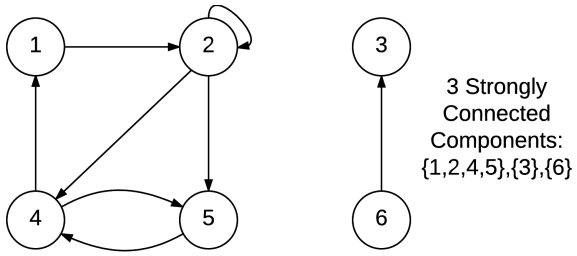
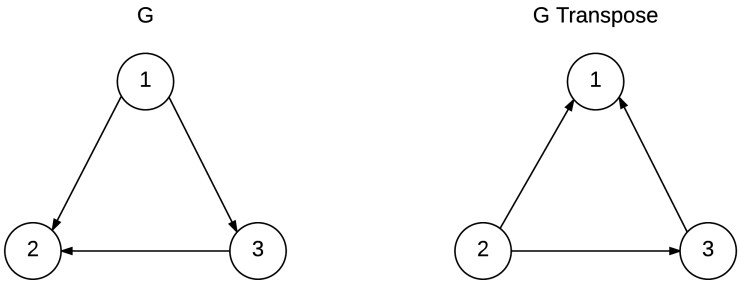


Figure 36: Example of a graph and its constituent strongly connected components

Given a graph G=(V,E) the transpose $G^T = (V, E^T)$ where $E^T = \{(u, v) : (v, u) \in E\}$



In order to find, algorithmically, the strongly connected components of a graph, execute the following steps:

1. Execute the DFS algorithm while also calculating for all u in V u.e.
2. compute G^T.
3. Execute DFS on G^T such that the nodes are considered in decreasing order of finish time computed in step 1.
4. Output each tree in the DFS forest calculated in step 3 as a SCC.

17.1 SCC Example

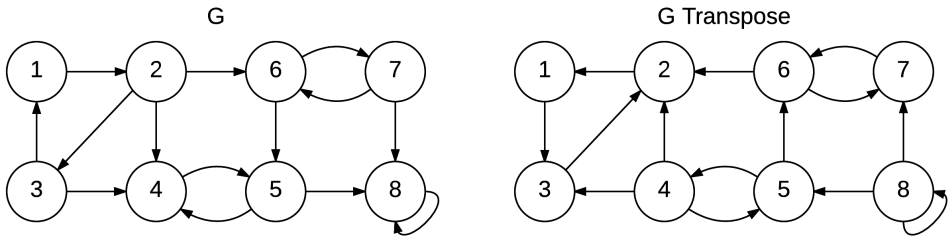


Figure 37: Graph and its transpose in which the SCCs will be found

1. Execute the DFS algorithm while also calculating for all u in V u.e.

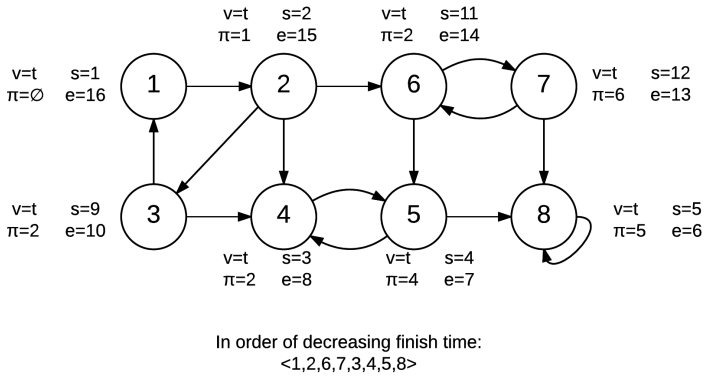


Figure 38: Execution of the modified DFS algorithm on G

2. compute G^T. This is given above.

3. Execute DFS on G^T such that the nodes are considered in decreasing order of finish time computed in step 1.

4. Output each tree in the DFS forest calculated in step 3 as a SCC. Each tree in the following forest represents a strongly connected component in the original graph.

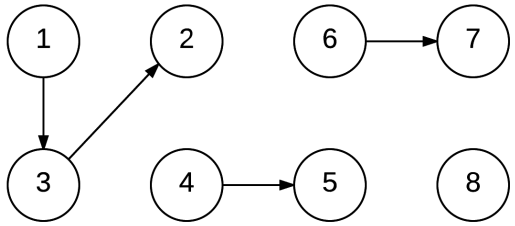
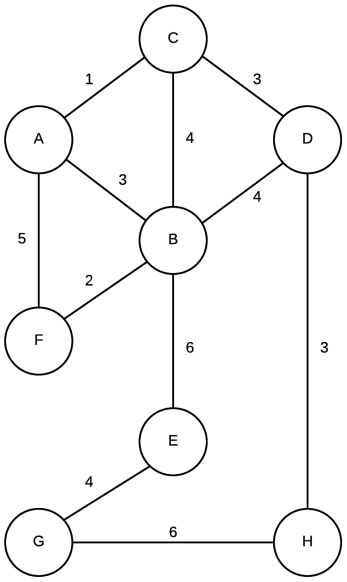


Figure 39: DFS forest resulting from the execution of the SCC finding algorithm

It can now be clearly seen that there exists 4 strongly connected components in the original graph: {1,3,2}, {6,7}, {4,5}, and {8}.

18 Minimum Spanning Trees

Problem There exists a set of servers which need to be connected with cables either directly or indirectly i.e. through multiple nodes, or hops. Cables have different cost, cost could be a function of length for example, and it is desired to find the minimal set of cables, and by extension cost, by which the entire set of servers may be connected. Represent the preceding as a



graph G(V,E) in which is vertex v is a server and the edges are the set of all possible cables which may connect servers. This will be a weighted graph. Functions:

- Weight function $w : E \to \mathbb{R}^+$ each edge is assigned a weight greater than zero
- Find $T^* \subseteq E$: it connects all servers and has minimal possible cost.
- Solve for $G^* = (V, T^*)$ such that the following are satisfied, given that p is a path:

$$\forall u, v \in V \exists p \subseteq T^*$$

$$\forall e \in T^* \left(\sum_{0}^{|T^*|} e \right) \text{ is minimal}$$

Once those conditions have been satisfied, the resulting graph G^* is the minimum spanning tree (MST)

19 Kruskal's Algorithm

Greedy MST Start from an empty set of edges - S=$\emptyset$. At each iteration add the edge in the graph with minimum cost which does not create cycles in S. Before the execution of the algorithm, no nodes are connected,

```
1 Kruskal(G,w)begin
2 |   S=∅
3 |   while ∃(u,v) ∈ E \ S : S ∪ {(u,v)} does not create cycles in S do
4 |   |   let D subseteqE \ S : each edge in D does not create cycles .if
  |   |    added to S
5 |   |   (u,v) = arg min_{(p,q)∈D} w(p,q)
6 |   |   S=S∪{(u,v)}
7 |   end
8 |   return S
9 end
```

this is denoted by the dotted borders around them. Also, no edges are selected, this too is denoted by the dotted lines. As the algorithm iterates, edges are chosen to connect nodes, and they are darkened, until no more edges need be chosen to connect any nodes, or as long as edges may be chosen which do not create cycles. In this example, edges were chosen in the following order, after which the graph is in the final state given below:S={(A,C),(F,B),(A,B),(C,D),(D,H),(G,E),(G,H)}

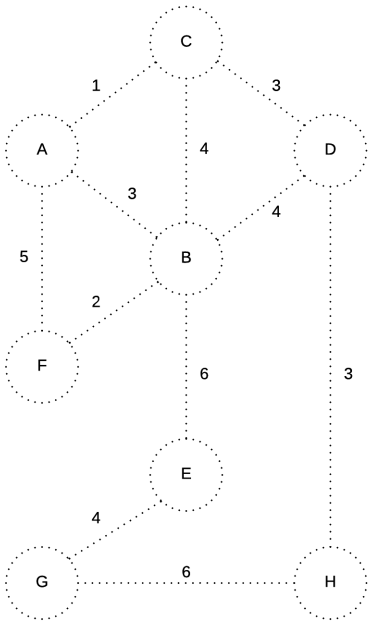


Figure 40: Before Kruskal's Algorithm execution

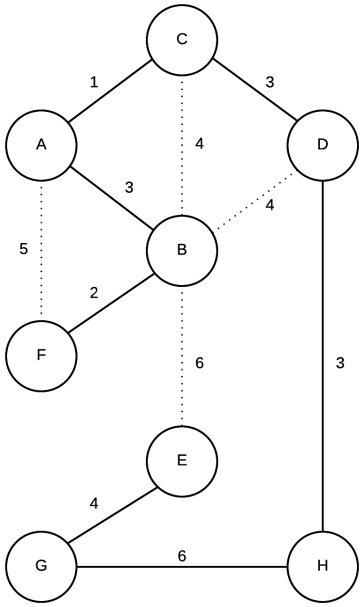


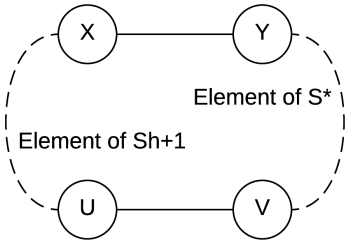
Figure 41: After Kruskal's Algorithm execution

19.1 Correctness of Kruskal's Algorithm

1. Termination: At each iteration, an edge is added to the solution and the loop terminates when a tree is generated. Therefore, at most, v-1 iterations will be done.

2. Induction: Let S_h be the solution at the h^{th} iteration.
   Prove that $\forall h \in \mathbb{Z}_n \exists$ an optimal solution $S^* : S_h \subseteq S^*$.

   - Base: $S_0 = \emptyset \rightarrow S_0 \subseteq S^*$
   - Assume: $\exists S^* : S_h \subseteq S^*$
   - Prove: $S_{h+1} \subseteq$ some optimal solution
     Let (u,v) be the edge selected at the h+1 iteration.
     If (u,v) $\in S^*$ then $S_{h+1} \subseteq S^* \rightarrow$ done.
     Else $S^* \cup \{(u, v)\}$ would have a cycle: (u,v) creates a cycle when



     added to S^* but does not create a cycle when added to S_h
     Therefore there exists at least one edge (x,y) in S^* which is not in S_h
     (x,y) in S^* does not create a cycle since $S_h \subseteq S^* \rightarrow$ the edge (x,y) does not create a cycle in S_h
     Therefor (x,y) was an edge which could have been selected by the algorithm at the h+1 iteration
     $S^{\#} = (S^* \setminus \{(x, y)\}) \cup \{(u, v)\}$
     $S^{\#}$ has n-1 edges
     $w(S^{\#}) \leq w(S^*)$
     $\therefore S^{\#}$ is optimal and $S_{h+1} \subseteq S^{\#}$
   - Final Solution is optimal
     The solution has n-1 edges
     There exists an optimal solution $S^* : S_{n-1} \subseteq S^*$

$|S^* = n - 1$
$|Sn - 1| = n - 1$
Therefore S_{n-1} is optimal

19.2 Efficient Implementation of Kruskal's Algorithm

- Sort edges in ascending order by weight
- The set of selected edges till the current iteration identifies a set of connected components
- An edge creates a cycle if it connects two nodes in the same component
- An array E is used for each edge i:
  - E[i].u and E[i].v are the endpoints
  - E[i].w is the weight of the edge
- An array CC[i...n] where n is the number of nodes in the graph - $|V|$. CC[i] is the id of the connected component of node i. Initially, $CC[i] = i \forall i \in [1 \dots n]$

19.3 Efficient Kruskal Implementation Pseudocode

The complexity of this algorithm is:

$$\Theta(m \log m + m + n^2 = \Theta(m \log m + n^2$$

The additive m is dropped in the simplification since m can be as much as n^2 and θ is concerned only with asymptotic behavior.

```
1  Kruskal(G,w)begin
2  |  S=∅
3  |  m=|E|
4  |  Sort E ascending by weight
5  |  for i=1 to n do
6  |  |  CC[i]=i
7  |  end
8  |  for j=1 to m do
9  |  |  (u.v)=(E[j].u,E[j].v)
10 |  |  if CC[u]≠CC[v] then
11 |  |  |  S=S∪{(u,v)}
12 |  |  |  cid=CC[v]
13 |  |  |  for p∈V do
14 |  |  |  |  if CC[p]==cid then
15 |  |  |  |  |  CC[p]=CC[u]
16 |  |  |  |  end
17 |  |  |  end
18 |  |  end
19 |  end
20 |  return S
21 end
```

20 Prim's Algorithm

Prim's is a greedy algorithm which also solves the minimum spanning tree, through from a slightly different logical approach to the problem. It has a root node, and at each iteration it extends the tree rooted at the root node by selecting the edge with minimum weight that connects a node in the tree with a node that is not yet in the tree. The algorithm terminates when all nodes belong to the tree, and that tree will be the minum spanning tree.

20.1 Prim's Algorithm Pseudocode

```
1  Prim(G,w)begin
2  |  S=∅
3  |  select a node r ∈ V as the root
4  |  C={r}//nodes currently in the tree
5  |  while C≠V do
6  |  |  (u, v) = arg min_{(x,y)∈E: x∈C∧y∋C}  w(x, y)
7  |  |  S=S∪{(u,v)}
8  |  |  C=C∪{v}
9  |  end
10 |  return S
11 end
```

It can be demonstrated that for any graph with only positive weights, both Prim's and Kruskal's algorithms will yield the same tree. Students may find it instructive to work the same graph problem with both algorithms to solidify this fact, and the differences in their logical execution.

20.2 Efficient Implementation of Prim's Algorithm

```
1  Prim(G,W,r)begin
2      for v ∈ V do
3          v.π=NULL
4          v.d=∞
5      end
6      r.d=0
7      Q=V //set of unvisited nodes
8      while Q≠ ∅ do
9          v=Q.ExtractMin()
10         for u∈ adj(v) do
11             if u ∈ Q ∧ u.d > (v, u) then
12                 u.π=v
13                 u.d=w(v,u)
14             end
15         end
16     end
17 end
```

Algorithm 27: Pseudo Code for an efficient implementation of Prim's Algorithm

Iterating through the algorithm, given the same graph used earlier in discussion of Kruskal (Figure 41), yields the following table of choices enumerated over the loop iterations, given that node A is specified as the root.

	Q=	A	B	C	D	E	F	G	H
	init	0	∞	∞	∞	∞	∞	∞	∞
1	Select A	~~0~~	3	1	∞	∞	5	∞	∞
2	Select C	~~0~~	3	~~1~~	3	∞	5	∞	∞
3	Select B	~~0~~	~~3~~	~~1~~	3	6	2	∞	∞
4	Select F	~~0~~	~~3~~	~~1~~	3	6	~~2~~	∞	∞
5	Select D	~~0~~	~~3~~	~~1~~	~~3~~	6	~~2~~	∞	3
6	Select H	~~0~~	~~3~~	~~1~~	~~3~~	6	~~2~~	6	~~3~~
7	Select G	~~0~~	~~3~~	~~1~~	~~3~~	4	~~2~~	~~6~~	~~3~~
8	Select E	~~0~~	~~3~~	~~1~~	~~3~~	~~4~~	~~2~~	~~6~~	~~3~~

Figure 42: Iterative Execution of Prim's Algorithm given in a table

Take note that in step 7 the algorithm had a choice, two edges were available of equal weight. In this case, one was chosen arbitrarily, but if you were to follow the other path in the execution of the algorithm, it too would have yielded a different, but still optimal solution. The resulting graph of the minimum spanning tree, which is the same as the one yielded for this graph when Kruskal's algorithm was applied, is given below.

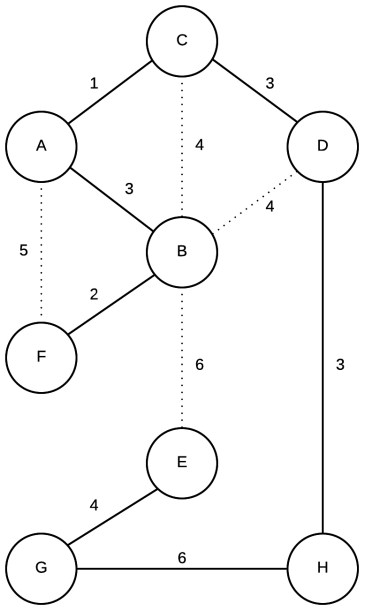


Figure 43: Graph of the Minimum Spanning Tree generated from Prim's Algorithm

21 Single Source Shortest Paths

There exists a network of servers connected by links, and each link has an associated delay. A router has to figure out the best path to the destination: the path with least delay. This can be modeled with a weighted graph G=(V,E) such that V represents the set of routers and E the set of links between routers with the weight function $w : E \to \mathbb{R}^+$, w(u,v) is the delay in the link $(u, v) \in E$.

Problem: Given a source $s \in V$ find the shortest path to all other nodes in the network.

Definition 21.1 (Path Weight)
Given a path $p = (v_1, v_2, \ldots v_k)$ the weight of p is the sum:

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

Definition 21.2 (Shortest Path)
The shortest path between two nodes u and v is defined as $\delta(u, v)$

$$\delta(u,v) = \begin{cases} \arg\min_{p:u \to v} w(p) & \textit{if } u \textit{ and } v \textit{ are connected} \\ \infty & \textit{otherwise} \end{cases}$$

21.1 Optimality Principle

Given two nodes u and v, their shortest path p, and a node q in their shortest path, then p', the path between u and q is also the shortest path for these two nodes.

21.2 Proof of the optimality principle by contradiction

- Assume p' is **not** the shortest path which exists between nodes u and q, and that p''' denotes the path from q to v.
- This implies that there exists another path p'' which is shorter.
- It follows that $w(p'') < w(p')$ and $w(p) = (w(p') + w(p''')) > w(p'') + w(p''')$
- This leads to a contradiction because p is not the shortest available path from u to v.

Combining all shortest paths from a source to all destinations is the shortest path tree, which is the goal of the following algorithms

22 Dijkstra's Algorithm for a shortest paths tree

- Greedy approach for calculating the shortest paths tree given a source node, for that node.
- Starts from the source node and extends the tree by adding nodes and corresponding edges.

- Let R be the set of nodes currently in the tree (up to the current iteration). The algorithm selects the edge (u,v) as follows:

$$(u, v) : e \in R \wedge v \ni R \wedge [\delta(u) + w(u, v)] \text{ is minimal}$$

- Attributes for a node u
  - u.d - is the distance from the source in the shortest paths tree. For the source, s, $s.d = 0$.
  - $u.d = \delta(s, u)$
  - $u.\pi$ is the parent of u in the shortest paths tree
  - R is the set of nodes in the tree.

```
1  DIJKSTRA(G,w,s)begin
2      s.d=0
3      s.π=NULL R={s}
4      while ∃(u,v) ∈ E : u ∈ R ∧ v ∋ R do
5          (u,v) = arg min_{(x,y)∈E: x∈R, y∋R} (x.d + w(x,y))
6          v.d=u.d+w(u,v)
7          v.π=u
8          R=R∪{v}
9      end
10 end
```

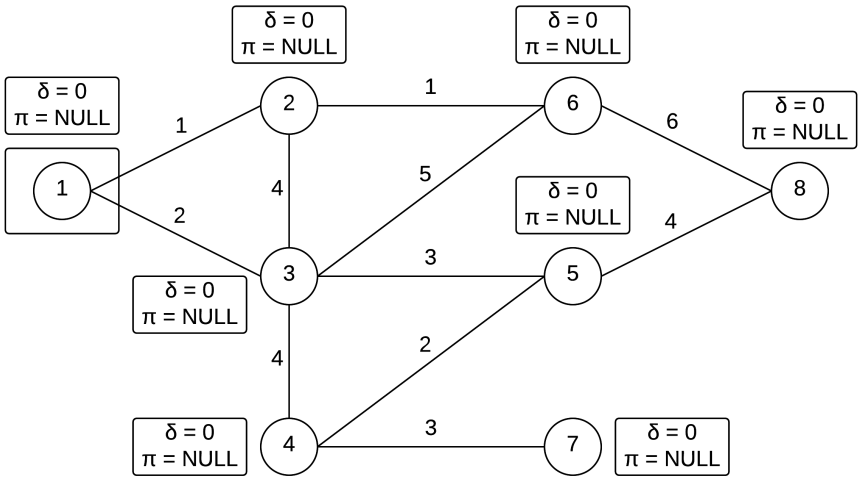


Figure 44: Initial state of graph

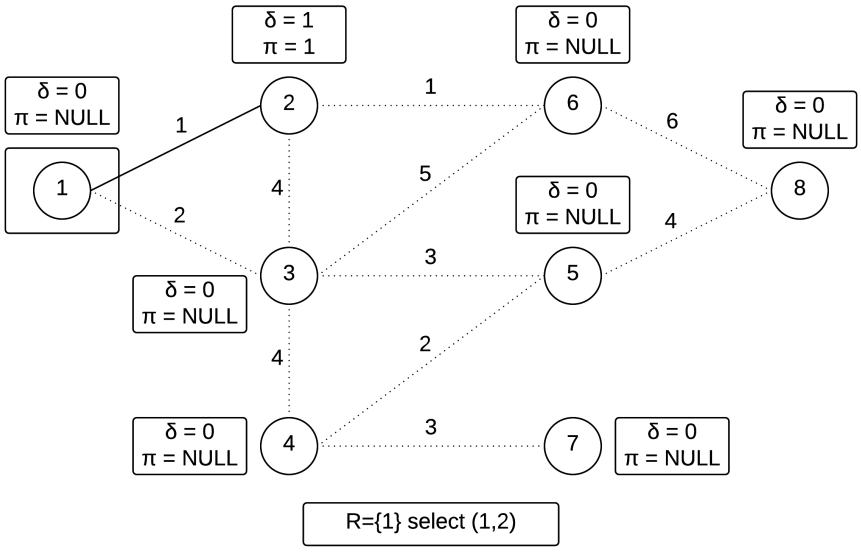


Figure 45: Step one of execution

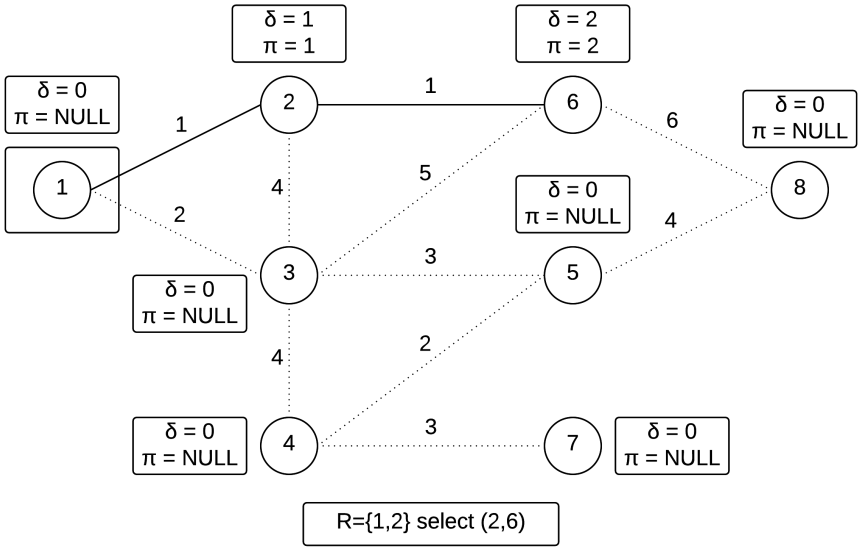


Figure 46: Step two of execution

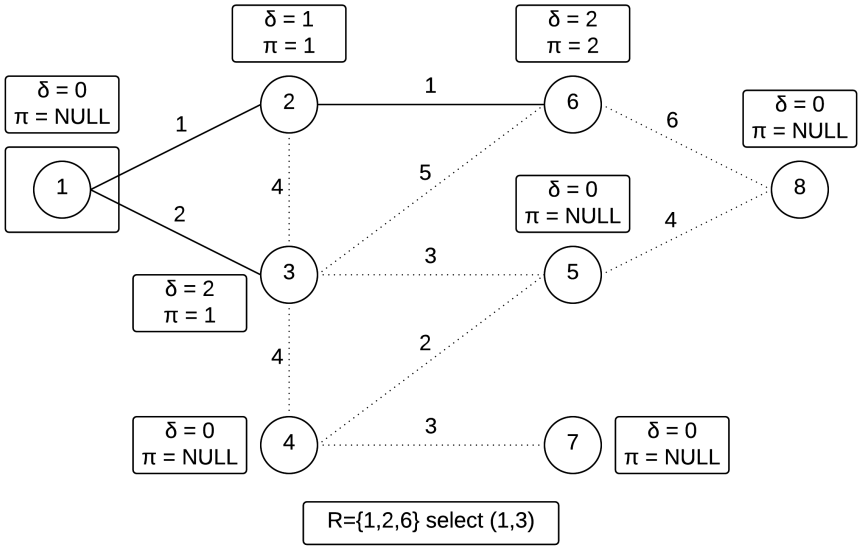


Figure 47: Step three of execution

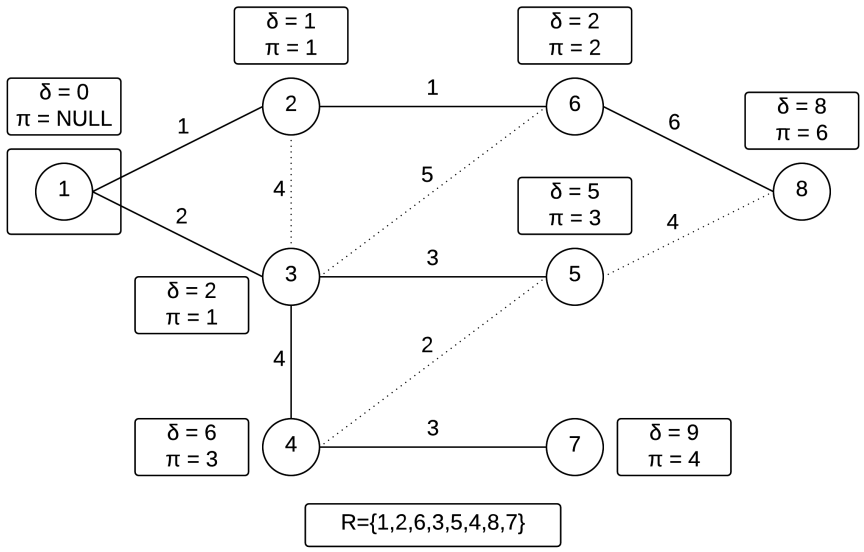


Figure 48: Final state

22.1 Efficient Implementation of Dijkstra's Algorithm

- Use a boolean array, to indicate $i \in R$

$$visited[1...n] : v[i] = \begin{cases} true & \text{if it the vertex has been visited by the algorithm} \\ false & otherwise \end{cases}$$

- integer k = number of visited nodes
- π - parent vector
- Integer array dist[1..n]:

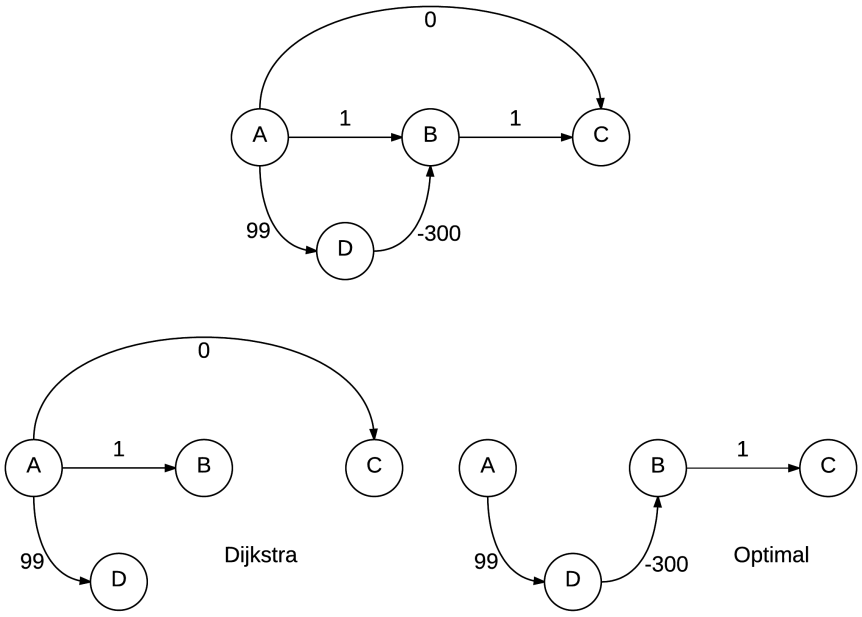
$$dist[i] = \begin{cases} 0 & i = source \\ \delta(s,v) & visited[i] \\ \text{current shortest path distance} & otherwise \end{cases}$$

Complexity=$\Theta(n^2 + m) \to \Theta(n^2) \because m \leq n^2$

```
1  DIJKSTRA(G,w,s)begin
2      dist[s]=0
3      π[s]=NULL
4      v[s]=True
5      k=1
6      for q ∈ V \ {s} do
7          v[q]=False
8          if q ∈ Adj(s) then
9              dist[q]=w(s,q)
10             π[q]=s
11         end
12         else
13             dist[q]=∞
14             π[q]=NULL
15         end
16     end
17     while k<n do
18         minD=∞
19         minV=NULL
20         for i=1 to n do
21             if v[i]==False∧dist[i]<minD then
22                 minD=dist[i]
23                 minV=i
24             end
25         end
26         v[minV]=True
27         k+=1
28         for q∈ Adj(minV) do
29             if !v[q]∧dist[q]>dist[minV]+w(minV,q) then
30                 dist[q]=dist[minV]+w9minV,q)
31                 π[q]=minV
32             end
33         end
34     end
35     return dist,π
36 end
```

23 Bellman-Ford

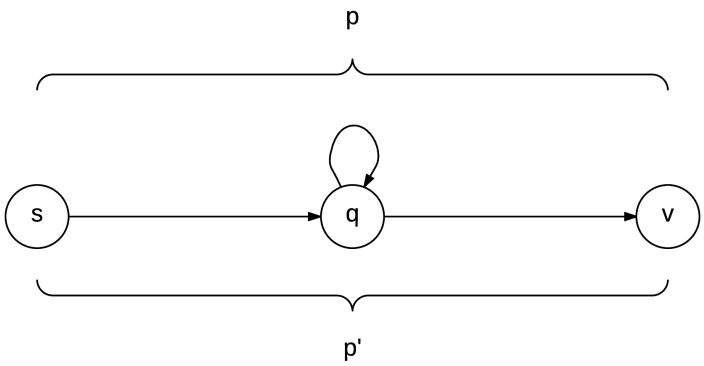
Consider the shortest path problem in which negative edge weights are permissible, such as the simple graph below.



Considering the shortest paths with negative weights adds other additional constraints:

- The problem makes sense iff there are no cycles with negative weight.
- The graph must be directed else even one negative edge may create this undesirable effect.
- Consider a graph G without negative cycles. If a node v is reachable from the source then the shortest path has length (number of hops) less than n.

Proof: Assuming there is a shortest path p with length $\geq$ n. This implies that there must exist in the path, a cycle. Iff that cycle is negative, then w(p)<w(p'). Without negative weighted edges, and cycles, this would be impossible.



23.1 Dynamic Programming Approach

- Define subproblems by considering shortest paths of at most length k:k=[0..n]
- Define a table M, of size k×v
- M[k,v] is the weight of the shortest path between the source s and v of at most length k if there exists a path of length less than or equal to k. M[k,v] equals infinity otherwise.
- Base case:k=0 → M[0,s]=0 and M[0,v]=$\infty \forall v \neq s$
- In general, for all k and v:

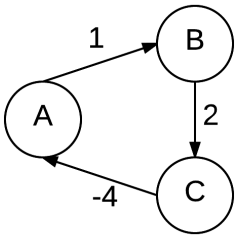
$$M[k,v] = \begin{cases} M[k-1,v] & \nexists \text{ shorter path} \\ M[k-1,u] + w(u,v) \forall u \in adj(v) & \exists \text{ shorter path} \end{cases}$$

- M[k,v]=min(M[k-1,v],min(M[k-1,u]+w(u,v):(u,v)∈ E))
- If $\exists v : M[n,v] \neq M[n-1,v] \rightarrow \exists$ negative cycles

Referring to the earlier graph for which Dijkstra's algorithm yielded a non-optimal solution, consider the following table illustrating the execution of the Bellman-Ford algorithm. Consider the following graph which is simple, but obviously contains a negative cycle and the following table illustrating the change earlier asserted as indicative of the presence of a negative cycle.

	A	B	C	D
0	0	∞	∞	∞
1	0	1	0	99
2	0	-201	0	99
3	0	-201	-200	99
4	0	-201	-200	99

Figure 49: Table for the Bellman-Ford execution on the earlier graph example opening this section



	A	B	C
0	0	∞	∞
1	0	1	∞
2	0	1	3
3	**-1**		

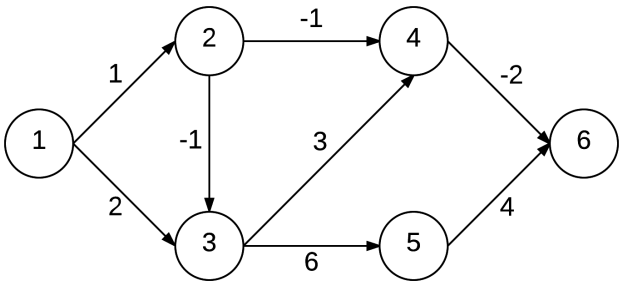
23.2 Bellman-Ford Pseudocode

Complexity of this algorithm is $\Theta(n(n+m)) \rightarrow \Theta(n^2)$

```
1  BellmanFord(G,w,s)begin
2      M[n+1[n]=new solution table
3      π[n]=new parent vector
4      ∀v ∈ V M[0, v] = ∞
5      M[0,s]=0
6      π[s]=NULL
7      for k=1 to n do
8          for v in V do
9              M[k,v]=M[k-1,v]
10             for (u,v)∈E do
11                 if M[k-1,v]+w(u,v)<M[k,v] then
12                     M[k,v]=M[k-1,v]+w(u,v)
13                     π[v]=u
14                 end
15             end
16             if (k==n∧M[k,v]≠M[k-1,v] then
17                 return "G has a negative cycle"
18             end
19         end
20     end
21     return M, π
22 end
```

23.3 Bellman-Ford Example Problem

Consider the following graph and construct the table generated during the execution of the Bellman-Ford Algorithm:



	1	2	3	4	5	6	
0	0	∞	∞	∞	∞	∞	
1	0	1	2	∞	∞	∞	
2	0	1	0	0	8	∞	
3	0	1	0	0	6	-2	*
4	0	1	0	0	6	-2	*
5	0	1	0	0	6	-2	
6	0	1	0	0	6	-2	

π	1	2	3	4	5	6
	NULL	1	2	2	3	4

Figure 50: Complete Bellman-Ford Execution Example

Note starred rows in the preceding example. In the context of this algorithm's execution, two consecutive row iterations that do not exhibit any changes indicate that the shortest paths have already been found. Once two rows remain unchanged, there will be no further changes in the table. Algorithm execution could halt at that point.

24 Floyd Worshall Algorithm

- Purpose of this algorithm is to calculate the shortest part between all pairs of nodes in a graph.
- Works with negative cycles: cycles having a negative weight.
- Dynamic Programming solution:

- consider the subproblems of calculating the shortest paths by considering the first k nodes in the graph $\{1 \dots \text{k}\}$. When k=n, the algorithm is complete.
- The graph is represented as an adjacency matrix W n×n:

$$W[i,j] = \begin{cases} 0 & i = j \\ w(i,j) & i \neq j \wedge (i,j) \in E \\ \infty & otherwise \end{cases}$$

Definition 24.1 (Intermediate Node)
Given a path $p = \{v_1, v_2, \dots v_l\}$ *an intermediate node of p is any node* $\in \{v_2, \dots v_{l-1}\}$ *i.e. not the first or last node.*

Consider the verticies $\{1 \dots k\} \forall k \geq 0$ for a pair of nodes $i, j \in V$, let p be the shortest path such that all intermediate verticies of p are elements in $\{1, \dots k\}$

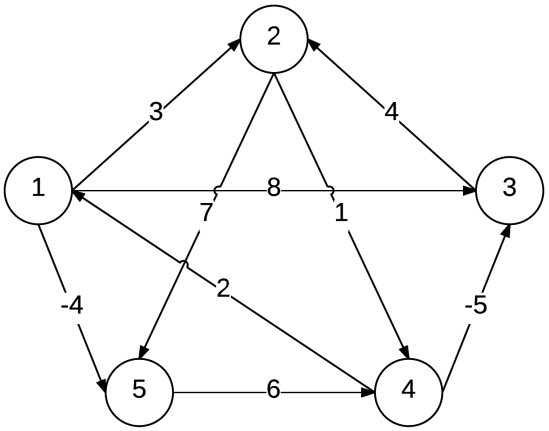
- If k is not an intermediate vertex or node of p, p is also the shortest path with intermediate nodes $[1 \dots k-1]$.
- If k is an intermediate vertex of p:
  - p_1 cannot have k as an intermediate vertex
  - p_1 and p_2 are both shortest paths due to the optimality principle
  - p_1 is the shortest path from i to k with {i+1...k-1} intermediate nodes .
  - p_2 is the shortest path between k and j {k+1...j-1} intermediate nodes.
- Let $d_{ij}^{(k)}$ be the weight of the shortest path between i and j with intermediate nodes {1...k}:

$$d_{ij}^{(k)} = min\left(d_{ij}^{(k-1)}, (d_{ik}^{(k-1)} + d_{kj}^{(k-1)})\right)$$

```
1  FloydWorshall(W,n)begin
2      D^(0) = W// n_0 zero intermediate nodes, direct edges only
3      for k=1 to n do
4          Let D^(k) be a new n×n matrix
5          for i=1 to n do
6              for j=1 to n do
7                  D_ij^(k) = min(D_ij^(k-1), (D_ik^(k-1) + D_kj^(k-1)))
8              end
9          end
10     end
11     return D^(n)
12 end
```

24.1 Floyd Worshall Example

Consider the following graph, and the tables which are generated from the algorithm execution.



$D^{(0)}$	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

Figure 51: FloydWorshall 0 - Direct edges - 0 intermediate nodes

$D^{(1)}$	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

Figure 52: FloydWorshall 1 - 1 intermediate node - $4 \rightarrow 1 \rightarrow 2$ and $4 \rightarrow 1 \rightarrow 5$ accounted for

$D^{(2)}$	1	2	3	4	5
1	0	3	8	$\underline{4}$	-4
2	∞	0	∞	1	7
3	∞	4	0	$\underline{5}$	$\underline{11}$
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

$D^{(3)}$	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	$\underline{-1}$	-5	0	-2
5	∞	∞	∞	6	0

Figure 53: FloydWorshall 2 and 3 (underlined values are those which have changed in the current iteration

$D^{(4)}$	1	2	3	4	5
1	0	3	$\underline{-1}$	4	-4
2	$\underline{3}$	0	$\underline{-4}$	1	$\underline{-1}$
3	$\underline{7}$	4	0	5	$\underline{3}$
4	2	-1	-5	0	-2
5	$\underline{8}$	$\underline{5}$	$\underline{1}$	6	0

$D^{(5)}$	1	2	3	4	5
1	0	$\underline{1}$	$\underline{-3}$	$\underline{2}$	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

Figure 54: FloydWorshall 4 and 5 (underlined values are those which have changed in the current iteration

25 Maximum Flow and Flow Networks

In the context of a graph, there exists two particular nodes, a source s and a destination t, such that the source generates data that are transported to the destination. The network can be represented as a directed graph where each edge has a capacity representing the amount of data that can go through the links. Intermediate nodes act as a relay and forward the data. This implies that, at each node, data in=data out i.e. no loss or gain.

MaxFlow problem: Find the maximum amount of data that the source can produce and can be sent to the destination through the network without violating capacity constraints. Consider the simple example below:

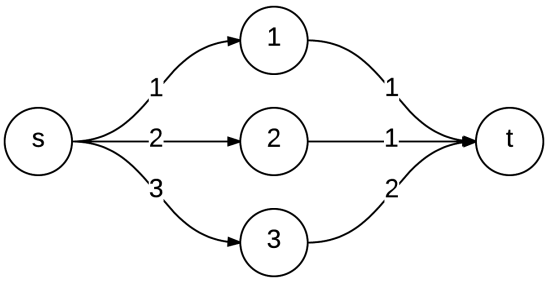


Figure 55: Simple Flow Network example in which the maxflow is 4

Definition 25.1 (Flow Network) *Directed grapg G=(V,E)*
$\forall (u,v) \in E c(u,v) \geq 0$ *is the capacity*
if $(u,v) \in E$ *then* $(v,u) \ni E$
No self loops
if $(u,v) \ni Ev) = 0$
$\exists$ *2 special nodes s and* $t \in V$
$\forall v \in V \setminus \{s,t\} \exists path : \{s, \ldots, v, \ldots, t\}$

Definition 25.2 (A flow) *A flow in G is a function* $f : V \times V \to R$ *:*

- $\forall (u,v) \in V, 0 \leq f(u,v) \leq c(u,v)$ *This is the capacity constraint*
- $\forall u \in V \setminus \{s,t\} \sum_{v \in V} f(u,v) = \sum_{v \in V} f(v,u)$ *Flow conservation constraint*

Definition 25.3 (Flow magnitude) *Given a flow of f the value $|f|$ is defined as follows:*

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Definition 25.4 (Residual Networks) *Given a flow network G and a flow f the residual network G_f is as follows:*

- *Same nodes as G*
- *$\forall (u, v) \in E$ residual capacity is:*
  - $c_f(u, v) = c(u, v) - f(u, v)$
  - $c_f(v, u) = f(u, v)$
- *In general:*

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & (u, v) \in E \\ f(u, v) & (v, u) \in E \\ 0 & otherwise \end{cases}$$

- *the edges in $G_f, E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$*

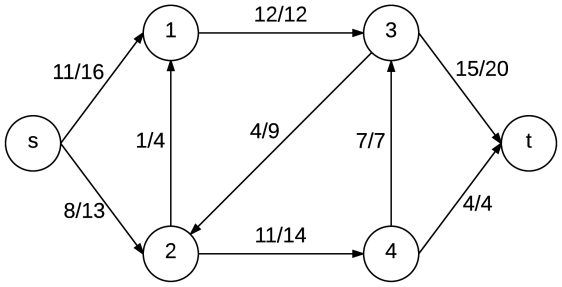


Figure 56: Flow Network

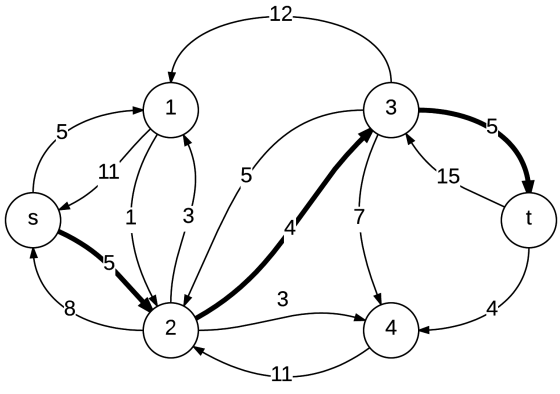


Figure 57: Residual Network with Augmenting Path in Bold

Definition 25.5 (Augmenting Path) *An augmenting path is a simple path from s to t in G_f. The maximum amount of flow that can be sent on an augmenting path p is defined as $c_f(p) = min\{c_f(u,v) : (u,v) \in p\}$. An example is shown in the preceding figure highlighted in bold. In that case the augmenting path has a maximum amount of flow equal to the maximum flow in the edge (2,3) which is 4.*

To find the Maxflow begin with the flow network and continue to add augmenting flows to the flow network, then generate the residual network. As long as more augmenting paths may be found, continue to iterate and add them to the flow. Once there exists no more augmenting paths in the residual network the algorithm much terminate.

Ford Fulkerson Algorithm Intuition

1. Start with an empty flow
2. Calculate the residual network
3. Find an augmenting path
4. Update the flow: add the path
5. Repeat steps 2-5 till no more paths exist

25.1 Ford-Fulkerson Algorithm Pseudocode

```
1  Ford-Fulkerson(G,s,t)begin
2      for (u, v) ∈ E do
3          (u, v).f = 0 //amount of flow through (u,v) initialized
4      end
5      calculate residual network G_f
6      while ∃p from s to t in G_f do
7          select an augmenting path p
8          c_f(p) = min{c_f(u, v) : (u, v) ∈ p}
9          for (u, v) ∈ p do
10             if (u, v) ∈ E then
11                 (u, v).f = (u, v).f + c_f(p)
12             end
13             else
14                 (v, u).f = (v, u).f − c_f(p)
15             end
16         end
17         calculate G_f
18     end
19     return //if the value of the maxflow is desired, sum all flow values
           leaving the source and subtract any entering
20 end
```

Complexity of Ford-Fulkerson If the capacities are all integer values, the algorithm is $O(E \times |F^*|)$ in the best case. Note: if the value of the maxflow is desired, sum all flow values leaving the source and subtract any entering.

25.2 Maxflow example

Consider the following graph and the graphs that follow which illustrate the execution of the Ford-Fulkerson Maxflow Algorithm:

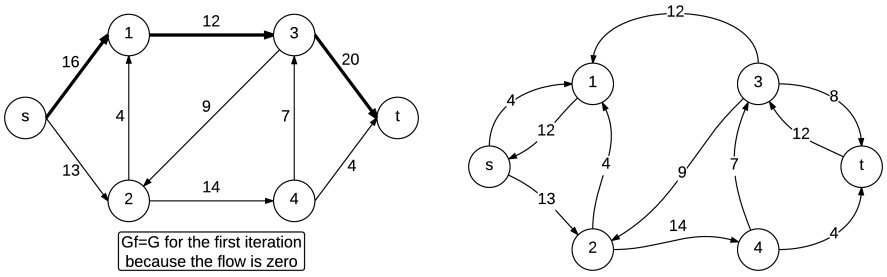


Figure 58: Ford-Fulkerson initial state and first iteration

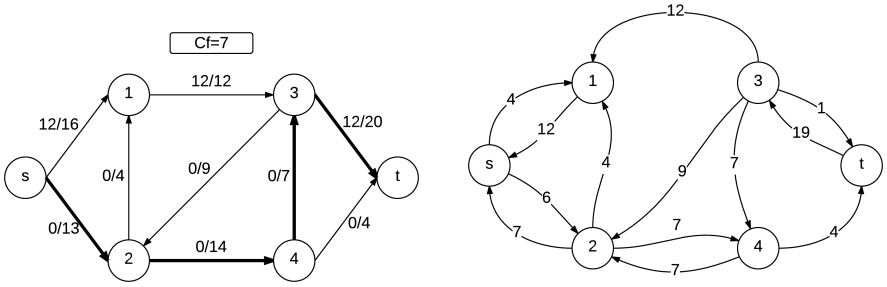


Figure 59: Ford-Fulkerson second iteration

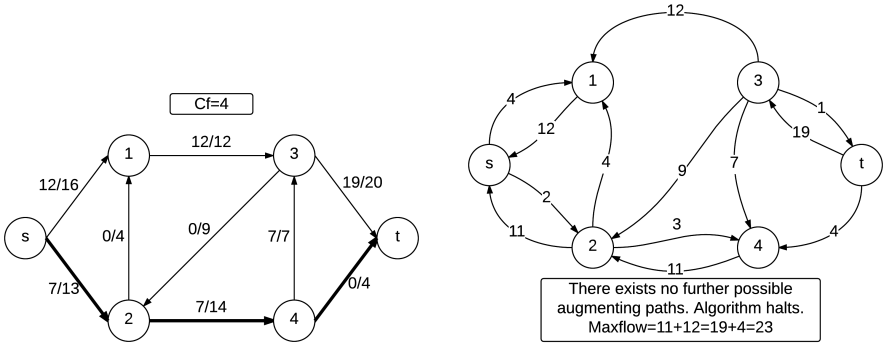


Figure 60: Ford-Fulkerson third iteration and halt

Definition 25.6 (Cuts) *A cut in a flow netwrok G+(V,E) is a partition of V into 2 sets S and T=V\S such that* $s \in S \wedge t \in T$.

Definition 25.7 (Cut Capacity) *The capacity of a cut (S,T) is c(S,T):*

$$c(S,T) = \sum_{\substack{(u,v)\in E: \\ u\in S \wedge v \in T}} c(u,v)$$

Definition 25.8 (Maxflow Min Cut Theorem) *Given a flow network G=(V,E) the value of the maximum flow for G is equal to the capacity of the minimum cut.*

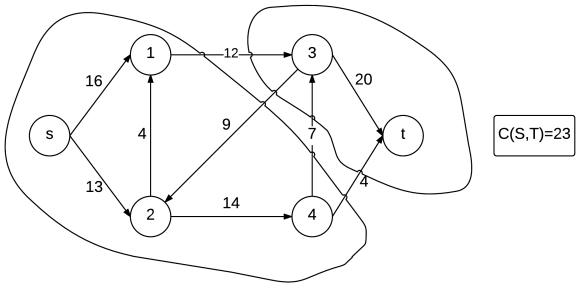


Figure 61: Cut in a graph illustrating the maxflow-min cut principle

25.3 Bipartite matching problem

For a group of employees, each employee, presumably, has a set of skills. There exists a set of tasks in which each requires a set of skill in order to be completed. The relationship between tasks and employees that may be involved in their completion is one to one. Find an assignment, matching, of employees to tasks that maximizes the number of tasks executed.

Modeling with a Bipartite Graph

Definition 25.9 (Bipartite Graph) *A Bipartite graph is a graph in which the set of nodes are partitioned into two set L and R such that the following holds:*

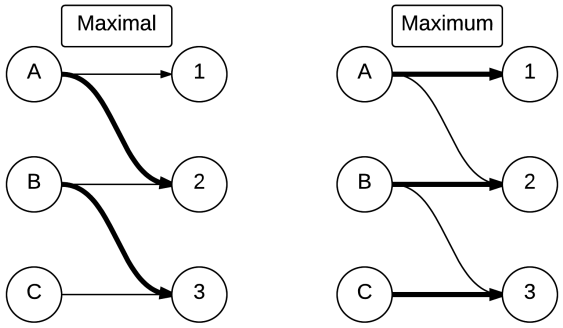
$$G = (L \cup R, E) \wedge L \cap R = \{\} \wedge \forall (u,v) \in E, u \in L \wedge v \in R$$

In the current problem, the set of employees would be assigned nodes in L and the set of tasks would be assigned to nodes in R. There would exist an edge (u,v) between an employee (a node u in L) and a task (a node v in R) the employee posses the necessary skills to perform the task. The set of edges E will then be a subset of all possible edges between all nodes in L and all nodes in R, or $E \subseteq L \times R$

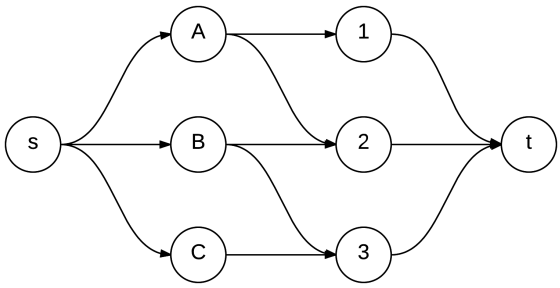
Definition 25.10 (Matching) *A matching M is a subset of E such that each node has at most one edge in the matching which is incident on that node.*

Definition 25.11 (Maximal Matching) *A matching M is maximal if it cannot be extended without violating one of the aforementioned rules.*

Definition 25.12 (Maximum Matching) *A maximum matching is an optimal solution.*



Solution The Ford Fulkerson Algorithm could be used to solve the problem by adding two nodes to the network, s and t, with an edge from s to each nodes in L and an edge from each node in R to t, as follows:



In this setup, the edges must become directed from the source to nodes in L and from nodes in L to nodes in R and from nodes in R to t. The Ford-Fulkerson Algorithm maximizes flow in the network, which here means number of completed tasks. The maxflow, because of the way in which the graph has been slightly restructured, corresponds to the max matching. This is the system parameter that was desired to be optimized. This procedure can be generalized to be applied in a wide array of settings and tasks, and optimization problems.