

# COMBINATORICS ALGORITHMS

GEORGE SALMAN

## CONTENTS

<b>Introduction.</b>	1
How we describe Algorithm as machine?	2
<u>“Hand-waving”</u> :	3
<b>Sorting Algorithms.</b>	4
Inserting sorting.	4
Inserting sorting.	4
Inserting sorting.	4
Merging Sort.	5
<u>Heap Sort</u> :	7
The choice problem.	9
<b>Seraching Algorithms in Graph.</b>	11
Ways to represent a graph:	12
BFS Algorithm.	12

## Introduction.

input: List.

Output: Sorted List.

What can we sort?

(1) Numbers.

(2) Game Cards.

(3) Planets in sun system as Distance from sun.

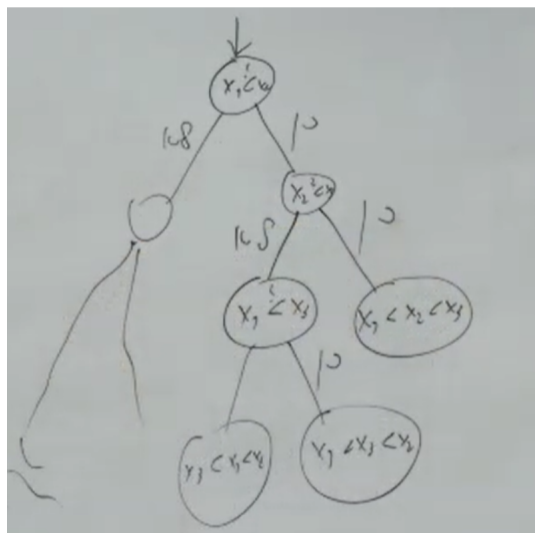
**Definition.** (The soting problem).

Given sequence of  $n$  real different numbers.  $a = (a_0, ..., a_{n-1})$ , the sorting problem is problem to find a unique permutation  $\pi \in S_n$  S.T

$$a_{\pi(0)} < a_{\pi(1)} < a_{\pi(2)} \dots < a_{\pi(n)}$$

When  $S_n$  is the group of permutations on  $\{1, 2, ..., n\}$  and  $\pi$  is called **Sorting permutation** of  $a$ .

### How we describe Algorithm as machine?



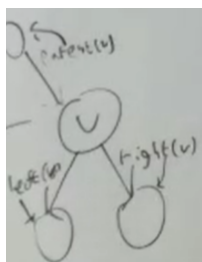
This is a full binary tree which mean in each vertice we have 0 child or 1 child or two child in this case it's a tree which has a significance i.e there is left and right child.

The big question is how many times we need to check equality between elements, so we can notice that in this graph the number of times we check is exactly the number of the edges, the path from the first vertice to the leaf is called the path length, and in case we are looking for the biggest length path it's called the tree height.

Now where  $n \log n$  come from?

notice that we have  $n!$  mix which we need to check so at least we have  $n!$  hence, we need to check in the case in which  $2^k > n! \rightarrow k > \log(n!)$  now the idea is that we need to look at the worst pasth in which we check all this mixes while the other pathes we check less but notice that in every level in the tree we get bigger in 2 times before i.e in level  $k \in \mathbb{N}$  we will have  $2^k$  the number of times required in sorting algorithm is when  $k > \log(n!)$ .

**Definition.** Binary tree  $T$  - is a directed graph (connected & no cycles) with root  $r$  s.t the out degree of every vertice is at most 2, the childs of every vertice  $v$  marked as  $left(v)$ ,  $right(v)$  if they exist, and the parent of  $v$  defined as  $parent(v)$  in case exist.



**Definition.** leaf in graph is vertice  $v$  with out degree 0 and the heigth of tree  $h(T)$  is length of the longest path from root  $r$  to leaf.

**Definition.** let  $n \in \mathbb{N}$  and let  $T$  be a full binary tree in which every vertice assuming  $v$  is mentions as  $q(v) = x_i ? x_j$  for  $0 \leq i, j < n - 1$  and every leaf mentioned by permutation  $\pi \in S_n$ , for all vector  $a = (a_0, \dots, a_n)$  we define a calculating path of  $a$  on a tree  $T$  as sequence  $v_0, \dots, v_m$  in inductive way when the base is  $v_0 = r$ , and step is assuming that  $v_k$  already defined, if  $v_k$  is leaf the sequence finished othrewise  $v_k$  is interior with query  $q(v_k) = x_i < x_j$ , and in this case,

$$v_{k+1} \triangleq \begin{cases} \text{right}(v_k), & a_i < a_j \\ \text{left}(v_k), & a_i \not< a_j \end{cases}$$

**Definition.** let  $T$  tree mentioned as in previous definition we will say that  $T$  is equality tree for sorting  $n$  elemnets if  $\forall a = (a_0, \dots, a_n) \in \mathbb{R}^n$  satisfied that the permutation which sort of  $a$  is equal to  $\pi(v_m)$  when  $v_m$  is the leaf in which get that calculation path of aon  $T$  and  $\pi(v_m)$  which written on this leaf.

*Claim.* every algorithm is based on equalities define for every  $0 < n \in \mathbb{N}$  the equalities for sorting  $n$  elements on  $T$  which describe it's operation on input of size  $n$ .

**Definition.** let  $A$  sorting algorithm which based on equalities algorithm complexity  $A$  is a function  $C_A : \mathbb{N} \rightarrow \mathbb{N}$  which for  $n$  return the max number of equalities in which  $A$  execute in running on  $n$  elements i.e  $C_A(n) = h(T)$  when  $T$  is the equality tree which  $A$  define on  $n$  elements.

*Claim.* A full binary tree in deepth of  $k$  at most obtain  $2^k$  leafs.

*Proof.* We will show the following in induction.

**Base:**  $k = 0$  we have  $2^0 = 1$  which is the tree root.

**Step:** assuming truth for  $k \in \mathbb{N}$  and we will show for  $k + 1$  i.e let  $T$  tree with depth  $k + 1$  we will remove from  $T$  the leafs and we get  $T'$  with depth  $k$  in  $T'$  we have from the induction the assumption that at most ewe have  $2^k$  leafs in tree  $T$  for every leaf we have at most 2 child h Since  $T$  is a binart tree hence, number of leafs is bouned by  $2 \cdot 2^k = 2^{k+1}$ .  $\square$

**“Hand-waving”:** Equality tree for sorting  $n$  elements should obtain  $n!$  leafs othrewise we have a  $\sigma \in S_n$  which didn't appear in the tree then we will return absouletely wrong answers on  $a = (\sigma(0), \dots, \sigma(n))$  hence, if  $T$  is equality tree which sort  $n$  elements then  $n! < 2^{h(T)}$  i.e  $lg(n!) \leq h(T)$  . now we can use Stirling's approximation in order to find approximation for  $n!$  which says  $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ , but we can approximate  $lg(n!)$  in other method.

First,

$$lg(x) = \frac{\ln(x)}{\ln(2)}$$

hence, it's enough to find low bounder for  $\ln(n!)$

$$\ln(n!) = \ln(1 \cdot 2 \cdots n) = \sum_{k=1}^n \ln(k) = \sum_{k=2}^n \ln(k) > \int_1^n \ln(x) dx = x \ln(x) - x \Big|_1^n$$

$$n \ln(n) - n - (\ln(1) - 1) = n \ln(n) - n + 1 = \Omega(n \ln(n))$$

*Remark.* \* stem from the definition of darboux sum in which we took the partition  $P = \{x_i\}_{i=1}^n$  s.t  $2 < x_1 < x_2 = x_1 + 1 < \dots x_n = n$ .

**Corollary.** *every sorting algorithm based on equalities executed in complexity of  $\Omega(n \log n)$ .*

## Sorting Algorithms.

### Inserting sorting.

**Inserting sorting.** This algorithm get list and take second elements at first and each time move to the next element in the array assuming  $i$  and compare it with elements before first with  $a[i-1]$  if  $a[i-1]$  smaller then we move to next otherwise we check if  $a[i-2]$  bigger then we keep that till we get  $a[m]$  in which  $a[m] < a[i]$ ,  $i < m$  then we swap  $a[m+1] \iff a[i]$  and the elements  $a[m+1] \dots a[i]$  go one step more E.g assuming we have  $a[5] = \{6, 7, 8, 7.5, 6.5\}$  and we want to use the algorithm first we have that  $7 > 6$  then we move to 8 and  $8 > 7$  so other step now we have 7.5 now  $7.5 < 8$  then we go step left and we see  $7.5 > 7$  so we swap it with 8 and all the element from 8 to 7.5 go step right so we have  $\{6, 7, 7.5, 8, 6.5\}$  now we look at 6.5 we see that till we get  $7 > 6.5$  we swap and move all the previous one step se we get  $\{6, 7, 7.5, 8, 6.5\} \rightarrow \{6, 6.5, 7, 7.5, 8\}$  and we are finished.

### Inserting sorting.

```
def insert_sort(a): ## a is list
    for k in range(1, len(a)):
        key = a[k]
        i = k - 1
        while(i >= 0 and a[i] > key):
            a[i+1] = a[i]
            i = i - 1
        a[i+1] = key
```

*Claim.* Denote the complexity of inserting sorting in  $C_{IS}(n)$  then  $C_{IS}(n) = \binom{n}{2} = n^2$ .

*Proof.* It's obvious that the algorithm  $IS$  in running the outside loop in line 2 (python code) executed exactly  $n - 1$  times  $\forall k \in \{1, 2, \dots, n - 1\}$  for every iteration  $k$  the equality in line 5 executed at most  $k$  times hence, the number of equalities is bounded by  $1 + 2 + 3 + \dots + n - 1 = \frac{n(n+1)}{2} = \binom{n}{2}$  this is upper bounder this upper bounder stem from choosing input E.g  $a = (n - 1, n - 2, \dots, n, 0) \in \mathbb{R}^n$  notice that 0 give worst complexity we can achieve in for condition  $i \geq 0$  and the other element to get worst number of equalities which is  $k$  in every iteration.  $\square$

**Merging Sort.** Given two sorted list assuming,

$$a = (a_0, \dots, a_{n-1})$$

$$b = (b_0, \dots, b_{m-1})$$

We need to find sorted  $C = (C_0, \dots, C_{n+m-1})$  in which it's elements are exactly  $a_0, \dots, a_{n-1}, b_0, \dots, b_{m-1}$ .

**Python Code.**

```

1  def merge(a,b):
2      n=len(a)
3      m=len(b)
4      i,j=0,0
5      c=[]
6      while i<n and j<m:
7          if a[i]<b[j]:
8              c.append(a[i])
9              i=i+1
10         else:
11             c.append(b[j])
12             j=j+1
13         if i==m:
14             while j<m:
15                 c.append(b[j])
16                 j=j+1
17         else:
18             while i<n:
19                 c.append(a[i])
20                 i=i+1
21     return c

```

when **Merge Sorting** algorithm look like:

```

1  def merge_sort(a):
2      n=len(a)
3      if n==0 or n==1:
4          return a
5      k=n//2 ##devide by 2 and give the low integer
6      b=merge_sort(a[:k])
7      c=merge_sort(a[k:])
8      return merge(b,c)

```

*Claim.* let  $C_m$  be the complexity of merge sort then  $C_m(n, m) \leq n + m - 1$  (i.e complexity of merge function).

*Proof.* Notice that merge do comparsion in line 7 as a part of loop in line 6, the loop start when  $i = 0, j = 0$  and finish when  $i == n \vee m == 0$  and in every itiration we increment  $i \vee j$  by 1, now look at  $i + j$ , in the beggining it's value is 0 and in every itiration  $i + j$  incremented by 1 hence, in the end itiration it's value at most is  $(n - 1) + (m - 1) = n + m - 2$  So the number of itiration is  $n + m - 1$  at most.  $\square$

**Theorem.** Denote in  $C_{ms}$  the complexity of merge sort then  $C_{ms} = O(n \log n)$ .

*Proof.* Notice that:

$$C_{ms}(n) \leq \underbrace{C_{ms}\left(\left\lfloor \frac{n}{2} \right\rfloor\right)}_{\text{line - 6}} + \underbrace{C_{ms}\left(\left\lceil \frac{n}{2} \right\rceil\right)}_{\text{line - 7}} + C_m\left(\left\lfloor \frac{n}{2} \right\rfloor, \left\lceil \frac{n}{2} \right\rceil\right)$$

Now we know the complexity of  $C_m$  hence,

$$\begin{aligned} \underbrace{C_{ms}\left(\left\lfloor \frac{n}{2} \right\rfloor\right)}_{\text{line - 6}} + \underbrace{C_{ms}\left(\left\lceil \frac{n}{2} \right\rceil\right)}_{\text{line - 7}} + C_m\left(\left\lfloor \frac{n}{2} \right\rfloor, \left\lceil \frac{n}{2} \right\rceil\right) &= \underbrace{C_{ms}\left(\left\lfloor \frac{n}{2} \right\rfloor\right)}_{\text{line - 6}} + \underbrace{C_{ms}\left(\left\lceil \frac{n}{2} \right\rceil\right)}_{\text{line - 7}} + \left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil - 1\right) \\ &= \underbrace{C_{ms}\left(\left\lfloor \frac{n}{2} \right\rfloor\right)}_{\text{line - 6}} + \underbrace{C_{ms}\left(\left\lceil \frac{n}{2} \right\rceil\right)}_{\text{line - 7}} + (n - 1) \leq 2 \underbrace{C_{ms}\left(\left\lceil \frac{n}{2} \right\rceil\right)}_{\text{line - 7}} + (n - 1) \end{aligned}$$

Now we will use the equality  $2 C_{ms}\left(\left\lceil \frac{n}{2} \right\rceil\right) + (n - 1)$  in order to show that:

$$\forall k \in \mathbb{N}, C_{ms}(2^k) \leq k \cdot 2^k.$$

**Base:**  $k = 0$ .  $C_{ms}(2^k) = C_{ms}(1) = 0 \leq k \cdot 2^k = 0$

**Step:**

$$C_{ms}(2^{k+1}) \leq 2C_{ms}(2^k) + (2^{k+1} - 1) \leq_{\text{induction}} 2 \cdot k \cdot 2^k + 2^{k+1} - 1 = *$$

$$* = 2^{k+1}(k + 1) - 1 \leq (k + 1)2^{k+1}$$

Now for some  $n$ . we choose  $k$  s.t

$$2^{k-1} < n \leq 2^k$$

i.e

$$2^k < 2n \Rightarrow k < \lg(2n) = \lg(n) + 1$$

Now

$$C_{ms}(n) \leq C_{ms}(2^k) \leq k \cdot 2^k < \lg(n+1) \cdot 2n = 2n \lg n + 2n = O(n \lg n).$$

□

*Remark.* Altogether with  $\Omega(n \lg n)$  which we saw for every sorting algorithm we deduce that merge sort complexity is  $\Theta(n \lg n)$ .

**Heap Sort:** We can look at array as tree of number like in the following photo:

```
def left(i):
    return 2i+1
def right(i):
    return 2i+2
def parent(i):
    return (i-1)//2
```

**Definition.** Heap is a binary tree which is able to comparison S.T  $\forall v$  which is not root,  $parent(v) > v$ , given array we can think of it as array with length  $n$  as a presentation of binary tree which is almost full in the following length:

```
def heapify(a,i,n):
    left_child = left(i)
    right_child = right(i)
    max_child_value = None
    max_child_index = None
    if left_child < n:
        max_child_value = a[left_child]
        max_child_index = left_child
    if right_child < n and max_child_value < a[right_child]:
        max_child_value = a[right_child]
        max_child_index = right_child
    if max_child_value is not None and a[i] < max_child_value:
        a[i], a[max_child_index] = max_child_value, a[i]
        heapify(a, max_child_index, n)
```

Algorithm which get a heap in which just the root could not satisfy the heap property and smaller than one of his sons-and give us a heap.

```
def create_heap(a):
    n=len(a)
    for i in range(n-1,-1,-1):
        heapify(a,i,n)
```

*Complexity of heapify:* We require in every running of heapify it execute at most two comparisons and maybe it call recursively itself for a subtree in which it's root is child of  $i$  hence, the complexity is has upper bounder by  $2 \cdot h(v_i)$  when  $v_i$  is a vertice from index  $i$  and  $h(v_i)$  is the tree depth which is it's root.

*Creating heap code in Python:*

```
1 def heap_sort(a):
2     h=len(a)
3     create_heap(a)
4     for i in range(1,n):
5         a[0],a[n-i]=a[n-i],a[0]
6         heapify(a,0,n-i)
```

*Claim.* The complexity of create\_heap is  $O(n)$ .

*Proof.* for some natural  $n$ , let  $k$  S.T  $2^{k-1} < n \leq 2^k$  Moreover, in a almost full binary tree with  $n$  vertices there is  $k$  levels  $0, 1, \dots, k-1$  and in the level  $i$  there is at most  $2^i$  vertices and it's length is  $k-i-1$  and the complexity of heapify  $\forall v_i$  bounded by  $2 \cdot h(v_i)$  hence,

$$\sum_{i=1}^n 2 \cdot h(v_i) \leq 2 \cdot \sum_{i=1}^n 2^i \cdot (k-i-1) \leq 2 \cdot \sum_{j=0}^k j \cdot 2^{k-j} = 2^{k+1} \cdot \sum_{j=0}^k \frac{j}{2^j}$$

Now we will bound  $\sum_{j=0}^k \frac{j}{2^j}$  S.T:

$$\sum_{j=0}^{\infty} j \cdot x^j = x \cdot \sum_{j=0}^{\infty} j \cdot x^{j-1} = x \cdot \left( \sum_{j=0}^{\infty} x^j \right)' = x \cdot \left( \frac{1}{1-x} \right)' = \frac{x}{(1-x)^2}$$

Now we plug in  $x = \frac{1}{2}$  and ge that:

$$\frac{\frac{1}{2}}{\left(\frac{1}{2}\right)^2} = 2$$



Hence,

$$2^{k+1} \cdot \sum_{j=0}^k \frac{j}{2^j} \leq 2^{k+1} \cdot 2 = 2^{k+2} = O(n)$$

□

Sorting using heap:

```

1  def heap_sort(a):
2      h=len(a)
3      create_heap(a)
4      for i in range(1,n):
5          a[0],a[n-i]=a[n-i],a[0]
6          heapify(a,0,n-i)

```

**Theorem.** The complexity of heap sorting is  $O(n \log n)$ .

*Proof.* The complexity is containing complexity of create.heap and  $n - 1$  times of heapify in complexity  $O(\log n)$  everyone hence,

$$C_{\text{heap-sort}}(n) \leq C_{\text{create-heap}}(n) + (n - 1) \cdot C_{\text{heapify}}(n)$$

$$\leq O(n) + (n - 1) \cdot O(\log n) \leq O(n \log n)$$

□

**The choice problem.**

**Definition.** given a list  $(a_0, a_1, \dots, a_n)$  with a sorting permutation  $\pi$  S.T  $a_{\pi(0)} \leq a_{\pi(1)} \leq \dots \leq a_{\pi(n)}$  we define  $\text{sel}(a, i) = a_{\pi(i)}$  in particular  $\min(a) = \text{sel}(a, 0)$ ,  $\max(a) = \text{sel}(a, n - 1)$ ,  $\text{median}(a) = \text{sel}(a, \lfloor \frac{n}{2} \rfloor)$ .

**Theorem.** We can calculate  $\text{sel}(a, k)$  in complexity of  $o(n)$ .

A naive algorithm to find median in a list.

```

18  def find_median(f):
19      insertion-sort(f)
20      return f[len(f)//2]

```

Code for the selection function:

```

def select(a,k):
    n=len(a)
    if n==1:
        return a[0]
    b=[find_median(a[5*i:5(i+1)]) for i in range (math.ceil(n/5))]
    x=select(b,len(b)//2)
    c=[a[i] for a[i] in a if a[i]<x]
    d=[a[i] for a[i] in a if a[i]>x]
    if len(c)==k:
        return x
    if len(c)>k:
        return select(c,k)
    if len(c)<k:
        return select(d,k-len(c)-1)

```

**Intuition.** In order to grasp the algorithm we will see some cases in which cover the whole idea, first we choose randomly element  $x$  in our list assuming  $a$  and assume without loss of the generality that we are looking for the element  $k$  in the list i.e  $select(a, k)$  first the algorithm till as to go on the list and check whether the chosen element bigger then  $x$  or less if it's bigger then we put it in the list  $C$  otherwise in  $D$  now if we have  $k$  elements in  $C$  then we are finished Since the  $k$  element should be  $x$  if it bigger than  $k$  we can apply the algorithm recursively on  $C$  till we find  $k$  and if it less then we search in  $D$  recursively.

**Remark.**  $b$  in the code represent the list of medians and notice that the element in  $b[\text{len}(b)//2]$  is the median of  $a$ .

**Claim.** if  $|a| = n$  then the size of the sets  $c, d$  in which the algorithm calculate is  $|c|, |d| \leq \frac{7}{10}n + 6$ .

**Proof.** This size of the set  $b$  is  $\lceil \frac{n}{5} \rceil$  and the number of element in  $b \leq$  the median  $x$  is  $\lceil \frac{b}{2} \rceil$  let  $F_1, F_2, \dots, F_t$  the set with order 5 in the partition of  $a$  so there is at least  $\lceil \frac{1}{2} \cdot \lceil \frac{n}{5} \rceil \rceil$  set between  $F_1, \dots, F_t$  in which the median  $\leq x$  every set like that give us 3 element in which they are less than  $x$  except the set of  $x$  and maybe one of the  $F$ 's which doesn't obtain 5 elements hence, the number of elements which are less than  $x$  is at least

$$|b| \geq 3 \left( \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq 3 \cdot \left( \frac{n}{10} - 2 \right) = \frac{3n}{10} - 6$$

this is the lower bound of the elements in which are less than  $x$  and from here the amount of elements which are bigger than  $x$  is at most  $n - (\frac{3n}{10} - 6) = \frac{7}{10}n + 6$  i.e  $|d| \leq \frac{7}{10}n + 6$  and identical explain tell us that  $|c| \leq \frac{7}{10}n + 6$ .  $\square$

Denote the complexity of  $C_{sel}(n)$  number of comparison in which *select* execute in the worst case of input with length  $n$  we will bound  $C_{Sel}$ . In step of finding  $b$  we execute a insertion sort to  $\lceil \frac{n}{5} \rceil$  sets of order 5 each one hence, it execute  $O(1)$  comparison for every set So in total we have  $O(n)$  comparison. Finding the median of  $b$  require complexity of  $C_{Sel}(\lceil \frac{n}{5} \rceil)$  Moreover, in the step of constructing  $c$  we compare all  $a$  elements to  $x$  i.e we do  $O(n)$  comparison and identical consideration

to  $d$ , and the recursive call run on  $c$  or on  $d$  in complexity of  $sel(\frac{7}{10}n + 6)$  so we got

$$\begin{aligned} C_{Sel}(n) &= O(n) + C_{sel}(\frac{n}{5}) + C_{Sel}(\frac{7}{10}n + 6) + O(n) + O(n) \\ &= O(n) + C_{sel}(\lceil \frac{n}{5} \rceil) + C_{Sel}(\frac{7}{10}n + 6) \end{aligned}$$

**Theorem.** exist  $t$  S.T  $C_{Sel}(n) \leq tn, \forall n > 0$ .

*Proof.* let a constant S.T the part in the formula of  $C_{sel}$  which is  $O(n)$  is bounded by  $an, \forall n \in \mathbb{N}$  now we define  $t = \max\{20a, C_{Sel}(140)\}$  We will show using induction on  $n$  that  $C_{sel}(n) \leq t \cdot n$ .

**Base:**  $1 \leq n \leq 140$  imply  $C_{sel}(n) \leq C_{sel}(140)$  using monotonic property hence, we conclude that  $C_{sel}(n) \leq C_{sel}(140) \leq t \leq tn$ .

**Assumption:** assume now on full induction that  $C_{sel}(k) \leq t \cdot k, \forall k < n$  and we will show to  $n$ . by the recursive formula which we already saw:

$$\begin{aligned} C_{Sel}(n) &\leq C_{sel}(\lceil \frac{n}{5} \rceil) + C_{Sel}(\frac{7n}{10} + 6) + a \cdot n \leq t \cdot \lceil \frac{n}{5} \rceil + t(\frac{7n}{10} + 6) + a \cdot n \\ &\leq t \cdot \frac{n}{5} + t + \frac{7tn}{10} + 6t + an \leq \frac{9t}{10} + 7t + an \leq t \cdot n + (\frac{-6n}{10} + 7t + an). \end{aligned}$$

hence, we get the required in condition that  $(\frac{-6n}{10} + 7t + an) \leq 0$  we extract  $t$  from the inequality and get that:

$$\begin{aligned} t \cdot (-\frac{n}{10} + 7) + an &\leq 0 \iff t(\frac{70 - n}{10}) \leq -an \\ \iff t(\frac{n - 70}{10}) &\geq an \iff t \geq \frac{10an}{n - 70} = 10a(\frac{n}{n - 70}) \end{aligned}$$

As remember we defined  $t = \max\{20a, C_{Sel}(140)\}$  now we can assume  $n \geq 140$  now  $\frac{n}{n-70}$  is decreasing function which is equal to 2 in  $n = 140$  hence,

$$10a \cdot (\frac{n}{n-70}) \leq 2 \cdot 10a = 20a \leq t$$

□

## Seraching Algorithms in Graph.

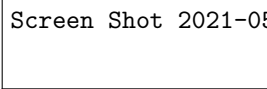
ABSTRACT. In this book we are going to discuss algorithms used in computer science, searching algorithms in graphs E.g sorting algorithms, BFS, DFS, and complexity and notations, every algorithm complexity truth is shown after so make sure there we no errors. those notes are accosiated by proff.Gadi Alexandrowitz at the math department and computer science technion.

**Definition.** Graph is a pair of vertices  $V$  and Edges  $E, G(V, E)$ .

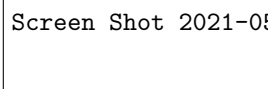
Screen Shot 2021-05-16 at 18.59.03.png

*Remark.* many of the objects in computer science involved are graphs. hence, there is significance use for algorithms in graphs E.g path searching, calculating length, and many graphs proberties. In the following course we will discuss undirected graphs (as represent in the follwing photo) and also for graphs in which every edge has direction. the arcs in directed graph will be represented by arrows.

### Ways to represent a graph:



- (1) neighbors lists: we hold a list of vertices when every element point to list

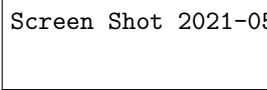


of it's neighbors as following:

- (2) Holding a matrix  $A$  s.t  $A_{ij} = 1 \iff (v_i, v_j) \in E$  i.e  $\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$ .

### BFS Algorithm.

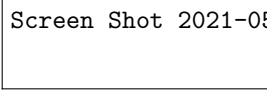
**(naive-form).** this is breadth running algorithm. we start with a vertice  $S$  and we run to all the neighbors which we can reach to.



**BFS Algorithm to find shortest pathes.** The algorithm will calculate two functions

- (1)  $d : V \rightarrow \mathbb{N} \cup \{\infty\}$  when  $d(v)$ = the length of the shrtest path from  $S$  to  $V$ .
- (2)  $\pi : V \rightarrow \mathbb{N} \cup \{NONE\}$  when  $\pi(v)$ = the vertice before  $v$  in the shortest path from  $S$  to  $V$ .

we will search it as fields  $v.d, v.\pi$ :



**Intiuition:** we start from  $S$  and breadthly run i.e we go to all the neighbors and we check there colors, for this child - this is the first time we reached so it's the shortest path to it. now a vertice which we reach and we still need to vistit it's neighbors marked by gray, and we mark by black to vertice which we reach already and visited it's neighbors.

### Algorithm Complexity:

- loop in line 4:  $O(|V|)$ .

- loop in line 10:  $O(|V|)$ .
- loop in line 10+12:  $O(|E|)$ .

In total we got  $O(|V| + |E|)$ .

**Algorithm correctness proof:** for every pair of vertices  $s, v \in V$  we denote  $\delta(s, v)$  the length of the shortest path from  $s$  to  $v$  or  $\infty$  if there is no such a path.  $\delta(s, v)$  is called the distance of  $v$  from  $s$  to  $v$  and a shortest path from  $s$  to  $v$  is a path from  $s$  to  $v$  with distance  $\delta(s, v)$ .

**Target 1:** to show in the end of the algorithm that  $d(v) = \delta(s, v)$ .

*Proof.* first we will show some lemas. □

**Lema 1 .** When the algorithm finish we have that  $\delta(s, v) \leq d(v)$ .

**Proof Lema 1.**

*Proof.* if  $d(v) = \infty$  the it's trivial.

now for vertice which we figured out while running the code, we will show the following in induction on the insertion step to  $Q$ .

**Base:** the base is for  $v = s$  in this case  $\delta(s, s) = 0 \leq d(s) = 0$  and indeed  $\delta(s, s) = 0 \leq 0$  (line 9).

**Step:** let vertice  $v \neq s$  which were inserted to  $Q$  while running the code in line 14 i.e exist a vertice  $u$  s.t the arc  $(u, v) \in E$  and we already detemined that  $v.d = u.d + 1$  since  $u$  were taken from  $Q$  in line 11 because  $u$  were inserted to  $Q$  before  $v$  therefore, by the induction assumption:  $\delta(s, u) \leq d(u)$  hence,  $\delta(s, v) \leq \delta(s, u) + \delta(u, v) \leq \delta(s, u) + 1 \leq d(u) + 1 = d(v)$  as required. □

**Lema 2.** if  $u$  were inserted to  $Q$  before  $v$  then  $d(u) \leq d(v)$ .

**Proof Lema 2.**

*Proof.* let  $v_0, v_1, \dots, v_t$  be sequence of vertices which were inserted to  $Q$  by there order we need to show that  $d(v_i) \leq d(v_{i+1}), \forall 0 \leq i \leq t$ .

if  $v_i = s$  then  $d(v_i) = 0$  (line-9) hence,  $d(v_i) \leq d(v_{i+1})$  automatically. therefore, from now we assume that  $v_i \neq s$  since,  $v_i \neq s$  exist  $u_i$  s.t  $v_i$  were inserted to  $Q$  in line-14 while scanning  $u_i$  and the same we will assume for  $v_{i+1}$  with vertice  $u_{i+1}$  now we split into cases:

1. if  $u_i = u_{i+1}$  then  $d(v_i) = d(v_{i+1}) = d(u_{i+1})$ .

2. if  $u_i \leq u_{i+1}$  then  $u_i$  were inserted to  $Q$  before  $u_{i+1}$ . otherwise scanning  $u_{i+1}$  before will lead us to the fact that  $v_{i+1}$  were inserted before  $v_i$  which is contradiction therefore, by the induction assumption we get that  $d(v_i) = d(u_i) + 1 \leq d(u_{i+1}) + 1 = d(v_{i+1})$ .

so we are finished. □

**Lema 3.**  $\delta(s, v) \geq d(v)$ .

**Proof Lema 3.**

*Proof.* we will show the following by induction on the size  $\delta(s, v)$ .

**Base:** if  $\delta(s, v) = 0$  then  $v = s$  hence,  $d(v) = d(s) = 0$ .

**Step:** let  $v \neq s$  s.t  $d(v, s) < \infty$ . we will look at the shortest path from  $s$  to  $v$ :  $s \rightarrow \dots \rightarrow u \rightarrow v$  when  $\delta(s, u) \leq \delta(s, v) - 1$  hence, by the induction step  $d(u) \leq \delta(s, v)$  moreover, the  $(u, v) \in E$  since,  $d(u) \leq \infty$  then  $u$  were inserted during running the code, scanned and were took out. while running the code, those are the options of  $v$  color:

1.  $v$  is white. then while scanning  $u$ , which were first detected and we will let  $d(v) = d(u) + 1$  hence,  $d(v) \leq d(u) + 1 \leq \delta(s, u) + 1 \leq \delta(s, v)$ .

2.  $v$  was gray. then exist a vertice  $w$  in which while scanning  $v$  were inserted to  $Q$  hence,  $d(v) = d(w) + 1$  since  $w$  were took out from  $Q$  while scanning which lead to insert  $v$  (line-11) then it was took out before  $u$  hence, it were inserted before  $u$  and from previous lema  $d(w) \leq d(u)$  hence,  $d(v) = d(w) + 1 \leq d(u) + 1 \leq \delta(s, v)$ .

3.  $v$  is black. therefore  $v$  were took out from  $Q$  before  $v$  hence, by previous lema we get that  $d(v) \leq d(u) \leq d(u) + 1 \leq \dots \leq \delta(s, v)$ .  $\square$

**Target 2:** In the finish of running **BFS** for every vertice  $v$  which we can reach from  $s$ , we need to show that the following path is the shortest path from  $s$  to  $v$ :  $s = \pi^{d(v)}(v) \rightarrow \pi^{d(v)-1}(v) \rightarrow \dots \rightarrow \pi(v) \rightarrow v$  and if we can't reach  $v$  then  $v.\pi = \text{None}$ .

*Proof.* if we can't reach  $v$  from  $s$  explicitly,  $\delta(s, v) = \infty$  as we saw  $\delta(s, v) = v.d$  in the finish of running the algorithm hence,  $v.d$  doesn't change while running the algorithm but if  $v.\pi$  was changed (line 17) then  $v.d$  was changed in (line-16), now we will assume that we can reach  $s$  from  $v$  and will show that following by induction on  $d(v)$ .

**Base:** if  $d(v) = 0$  then  $v = s$  and indeed  $s = \pi^0(s) = \pi^{d(v)}(v)$ .

**Step:** let soe  $v$  S.T  $d(v) < \infty$  (i.e we can reach  $v$  from  $s$ ) S.T  $v \neq s$  let  $u = \pi(v)$  this value were plugged in in  $v.\pi$  (line 17) hence, (line 16) determined that  $v.d = u.d + 1$  i.e  $d(u) = d(v) - 1$  therefore, we can use the induction assumption on  $u$ .

$$s = \pi^{d(v)-1}(u) \rightarrow \dots \rightarrow \pi(u) \rightarrow u$$

i.e

$$s = \pi^{d(v)-1}(u) \rightarrow \dots \rightarrow \pi(u) \rightarrow u$$

Moreover,  $u = \pi(v)$  i.e:

$$s = \pi^{d(v)-1}(\pi(v)) \rightarrow \dots \rightarrow \pi(\pi(v)) \rightarrow \pi(v)$$

we will add the arc  $u \rightarrow v$  in which it's existence insured by line 12 and we get:

$$s = \pi^{d(v)}(v) \rightarrow \dots \rightarrow \pi^2(v) \rightarrow \pi(v) \rightarrow v$$

$\square$

**Corollary.** In total we showed the truth of the field  $v.d$  and the truth of the field  $v.\pi$  hence, the algorithm **BFS** which we represented work as required.

*Intuition to BFS:.*

