DATA STRUCTURES
AND ALGORITHMS

DEPARTMENT OF COMPUTER SCIENCE
NJIT

ADDITIONAL INFORMAL LECTURE NOTES

by WOJCIECH RYTTER
Euclid algorithms, computing gcd.

gcd(a, b) - greatest common divisor of a and b. We have:

\[ n > m \implies gcd(a, b) = gcd(a - b, b) \& gcd(a, b) = gcd(a \mod b, b) \]

We use these equalities in two algorithms, the main invariant is that the greatest common divisor is the same.

**Algorithm Euclid1**

```
while a != b
    if (a > b) a = a - b; else b = b - a
return b.
```

**Algorithm Euclid2**

```
while a != b
    if (a > b) a = a \mod b; else b = b \mod a
return b.
```

Let \( n \), size of the input, be the total number of bits needed to write \( a \) and \( b \). The complexity of Euclid1 is \( \Theta(2^n) \), since for \( a = 2^{n-1}, b = 1 \) it makes \( 2^{n-1} \) tests \( a! = b \). The size of the number \( a \) is \( n \).

The complexity of Euclid2 is \( \Theta(n) \), since after two iterations the larger number decreases at least by half.

Justification:

**Case 1:** \( b \leq a/2 \), then \( a \mod b \leq b \leq a/2. \)

**Case 2:** \( (b > a/2) \), In the next iteration \( a' < a - b \leq a - a/2 = a/2 \)

Since the size is halved after at most two iterations the total number of iterations is proportional to the number of bits, so it is linear.

Both algorithms work in the same way if \( a, b \) are two consecutive Fibonacci numbers: 1, 2, 3, 5, 8, 13, 21, 34, 55.

Assume \( a = 55, b = 34 \). Then consecutive values of the pair \((b, a)\) are:

\[(34, 55) \rightarrow (21, 34) \rightarrow (13, 21) \rightarrow (8, 13) \rightarrow (5, 8) \rightarrow (3, 5) \rightarrow (2, 3) \rightarrow (2, 1) \rightarrow (1, 1)\]
The majority problem


find an element (leader) which appears more than n/2
or report there is no such an element

Assume initially the sequence A is sorted, we need to find the longest "run"
sequence of the same elements which are consecutive
Sorting need O(n log n) time, not linear altogether

Algorithm:
max=1; leader=A[0];

for (i=2;i<n;i++) if A[i]==A[i-max]
{max++; leader=A[i];}

Assume now the sequence is not necessarily sorted.
Another strategy:
Assume there is a leader. After removing any two distinct elements
the leader does not change.

Algorithm:
while all elements are not the same do
remove any two distinct elements

The sequence can be implemented as 2-way list, we scan
the list from left to right until two different elements are
found, then two elements are removed. The total time is linear.

However we do not need to use lists, the working sequence would
consist of an element A[j] repeated mult times, and the subsequence
Three additional simple variables are enough int i, j, mult. Assume there is a leader. At the end we have to verify if the computed leader is really a leader.

Algorithm
// in place, linear time, small extra memory

j=0; mult=1;

for (i=1;i<n;i++) {
    if mult==0 {mult++;j=i};
    else --mult;
}

if mult>0 return A[j] else "no leader"

The algorithm makes n-1 tests A[i]==A[j] additionally n-1 are needed to verify the leader
Other interesting fast and simple algorithms:

1. Given an array \( A[0..n] \) of integers, for each position \( i \leq n \) compute in linear time the first position \( k (k = LEFT[i]) \) to the left with \( A[k] > A[i] \), if there is such a position. If there is no such position the \( LEFT[i] = -1 \).

2. Given two sorted arrays of integers \( A[1..n], B[1..n] \), Compute in \( O(\log n) \) time the median of the join sequence \( A \cup B \). Assume for simplicity all elements are distinct.

3. Given an array \( A[1..n] \) of integers which is ”almost sorted”: each element is at a constant distance from its location in a sorted sequence. Sort \( A \) in linear time (insertion sort)

4. Show that the number of inversions in a permutation is of a same order as insertion sort for this permutation

5. Compute number of inversions in \( O(n \log n) \) time (modify Merge-Sort).

6. Given two sorted arrays of integers \( A[1..n], B[1..n] \). Compute in linear time if there are \( a \in A, b \in B \) such that \( x = a + b \), for a given \( x \).

7. Given array of positive integers change in linear time and in-place to a sorting table (with respect to modulo 3), by swapping elements. It is called also Dutch flag problems, three colors, sort colored elements so that colors are in good order.
The problem ”Stable marriages”

2D-array algorithm: stable marriage problem

Assume we have \( n \) women and \( n \) men. Each man has its list of preferences, on such a list each woman appears exactly once. The woman closer to the beginning of the list is more preferable. Similarly each woman has its preference list.

Example. Assume the list are written vertically, beginning from the top. The man are denoted by capital letters, the women correspond to numbers.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>E</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>A</td>
<td>E</td>
<td>D</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>D</td>
<td>B</td>
<td>B</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>C</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>A</td>
</tr>
</tbody>
</table>

A set of marriages is called unstable if two people who are not married both prefer each other o their spouses. For example the set:

\[
A_1 \ B_3 \ C_2 \ D_4 \ E_5
\]

is not stable because A prefers 2 to 1, and 2 prefers A to C. We can make \( \text{switch}(A,2) \) to get

\[
A_2 \ B_3 \ C_1 \ D_4 \ E_5
\]

However now B, 2 make an unstable pair. We can marry them and get

\[
A_3 \ B_2 \ C_1 \ D_4 \ E_5
\]

Now B and 1 make unstable pair, we marry them, we get

\[
A_3 \ B_1 \ C_2 \ D_4 \ E_5
\]

next A and 1 are unstable, after marrying them we get the configuration

\[
A_1 \ B_3 \ C_2 \ D_4 \ E_5
\]

from which we have started. Our algorithm is making a cycle. Hence such an algorithm does not make sense and a completely different approach is needed.
Let \( \text{rank}[w,k] \) be the rank of \( k \)-th man on the list of woman \( w \). Let \( \text{prefer}[s,i] \) be the \( i \)-th woman on the preference list of the man \( s \).

We add additional artificial man number 0, his priority is the lowest. We have
\[
\text{rank}[0] > \text{rank}[k] \text{ for each } 1 \leq k \leq n
\]
Assume that initially we have table next equal to zero everywhere, we have also the table fiancee\([w]\) initially set to zero for each woman \( w \).

**Stable marriage algorithm**

```plaintext
for (m=1;m<=n;m++){
    for (s=m;s!=0;){
        next[s]++;
        w=prefer[s,next[s]];
        if rank[w,s]<rank[w,fiancee[w]]
            {t=fiancee[w]; fiancee[w]=s; s=t;}
    }
}
```

The algorithm is stable because each woman to whom he proposes and is rejected has a better partner. Hence there is no way that any man can find a better partner which prefers him to her actual spouse.

The complexity of the algorithm is \( O(N) \), where \( N = n^2 \), it is linear with respect to the size \( N \) of the input.
Circular lists and Josephus problem:

Suppose \( n \) people are arranged in a circle and \( 1 \leq m \leq n \). Beginning with the person 1 we proceed around the circle and remove every \( m \)-th person. The order in which the elements are removed is called the \( (n, m) \)-Josephus permutation. The last removed element is denoted by \( J(n, m) \).

We can use circular lists to compute \( J(n, m) \), as well as to compute the Josephus permutation.

```c
struct node { int item; node* next;
    node(int x, node* t) {item=x;next=NULL;} }
typedef node *link;

void main()
{int i,n,m; cin>>n>>m;
  //initialize circular list with 1,2,...,n
  link t=new node(1); link x = t;
  for (i=2;i<=n;i++) {x->next=new node(i); x=x->next;}
  x->next=t;

  while (x!=x->next)// more than one element
    {
      for(i=1;i<m;i++) x=x->next;//find m-th element
      x->next=x->next->next; //remove m-th element
    }

cout<< x->item<<endl; }
```

Let \( n \) be a positive integer. Denote \( \text{shift}(n) \) the number which results from \( n \) by moving the leftmost 1 in the binary repr. of \( n \) after the last binary digit. For example

\[
\text{shift}(12) = \text{shift}([1100]_2) = [1001]_2 = 9.
\]

We have the following facts:

\[
J(2k) = 2 \times J(k) - 1; \quad J(2k + 1) = 2 \times J(k) + 1; \quad J(n,2) = \text{shift}(n).
\]

For example if 12 people are in the circle then the last survivor is 9.

It can be proved by induction w.r.t. \( n \). One has to consider two cases: \( n \) is even or \( n \) is odd. In other words what is the last binary digit of \( n \).
Lists: basic operations

Assume that link, node are defined as before (1-way list nodes). We list several simple functions for lists, assuming link points to the first element of a list or a sublist.

```
// reverse the list, iterative algorithm
link reverse(link x)
{  link t; y=x; r=0;
   while (y!=0) { t=y->next; y->next=r; r=y; y=t;}
   return r; }

// recursive reversing
link reverse(link x)
{if (x->next=0) return x;
   link t = reverse(x->next); x->next->next=x; x->next=0;
   return t; }

int count(link x){ if (x==0) return 0; return 1+count(x->next);}

// iterative traversal
void traverse(link h, void visit(link))
{ for link t=h; t!=0; t=t->next) visit(t);}

// recursive traversal
void traverse(link h, void visit(link))
{ if (h==0) return; visit(h); traverse(h->next,visit); }
\typical visit is: cout<<h->item;

// traverse in reverse order
void traverseR(link h, void visit(link))
{ if (h==0) return; traverseR(h->next,visit); visit(h); }

// remove all nodes containing a specified item
void remove(link& x, Item v)
{ while (x!=0 & x->item==v)
   {link t = x; x=x->next;delete t;}
   if (x!=0) remove(x->next,v); }
```
A decision tree for sorting 3 elements with at most 3 comparisons

the elements are 1, 2, 3

```
1 < 2 ?

yes

2 < 3 ?

1 < 3 ?

1 2 3

no

1 < 3 ?

2 1 3

3 < 2 ?

3 2 1

1 3 2

3 1 2

2 3 1
```
Sorting with small number of comparisons

Sorting 5 elements with at most 7 comparisons

PHASE 1:

after suitable 3 comparisons
we can get the following
schematic situation

PHASE 2: insert e into a < c < d with at most 2 comparisons

PHASE 3: there are two disjoint cases which depend on situation from PHASE 2

CASE 1:

if d < e then we get
at most two comparisons more
to insert b into a < c

CASE 2:

if e < d then we get
now insert b into sequence of
the first 3 elements
using at most 2 comparisons
Sorting with small number of comparisons

Denote by $S(n)$ the minimal worst-case number of comparisons of the best algorithm for sorting $n$ elements. We know that $S(n) \geq \lceil \log n! \rceil$, in some cases the equality holds. For most $n$ the exact value is not known.

For $n = 1, 2, \ldots, 11$ and $n = 20, 21$ Ford-Johnson algorithm achieves $S(n)$, hence it is optimal. The previous algorithm for $n = 5$ was a simple case of Ford-Johnson algorithm for $n = 5$.

The value of $S(13)$ has been computed in 2002, it was an open problem for about 30 years.

<table>
<thead>
<tr>
<th>$n$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S(n)$</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td>19</td>
<td>22</td>
<td>26</td>
<td>30</td>
<td>34</td>
<td>66</td>
</tr>
<tr>
<td>$\lceil \log n! \rceil$</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td>19</td>
<td>22</td>
<td>26</td>
<td>29</td>
<td>33</td>
<td>66</td>
</tr>
</tbody>
</table>

Ford Johnson algorithm

We show how it works for $n = 11$.

First arrive at situation as the one in figure below. The sequence $b_1, a_1, a_2, \ldots, a_5$ is called the main branch.

Insert the sequence $b_2, b_3, \ldots, b_6$ into the main branch by binary search in the order $b_3, b_2$, then $b_5, b_4$ then $b_6$. Altogether it takes 26 comparisons which is the same as $\lceil \log 11! \rceil$. Hence the algorithm is optimal for case $n = 11$.

For smaller $n$ the algorithm works in the same way, first crating the main branch and then inserting other elements into the main branch in the same order, if $n$ is smaller some elements are skipped as nonexistent.

The main branch is $b_1, a_1, a_2, a_3, \ldots$ Other elements are inserted in a special order into main branch using binary insertion.
Red-Black trees

Red-black trees. Binary search trees, the nodes are colored red and black (we draw red as white). Assume external leaves are added, two external leaves per each normal leaf, and one external leaf for each internal node with one child. External leaves are black. The number of blacks on each path from the root to each external leaf (including external leaf) is the same. Red node cannot have red parent. The root is black. The root is black.

We can omit external leaves in the definition by adding the (equivalent) condition: each internal node has two children or one child which is a red leaf. This property guarantees logarithmic height.

An example of a red-black tree:

![Red-Black Tree Diagram]

Basic properties of red-black trees:

1. the height is $O(\log n)$
2. insertion and restructuring can be done in $O(\log n)$ time, logarithmic number of recolouring, single operation tree node
3. deletion can be done in $O(\log n)$,
Red-Black trees: insertion

Newly inserted node $x$ is initially red, to satisfy that the number of blacks on each path is the same. The "bad" situation happens when $x$ becomes a child of red node. Generally we iterate the cases until there will be no "badness" (red child of red parent).

Two cases: the "uncle" approach. The uncle is the sibling of grandparent which is not the parent. Let $\text{parent}(x) = y$, $\text{parent}(y) = z$, $\text{uncle}(x) = u$. Apply one of three cases.

**Case 1:** (terminates insertion) there is no uncle. Make $x$ and $y$ children of $z$.

**Case 2:** the uncle of $x$ is red. We do the recoloring, $y$ and $u$ are colored black.

If $z$ is not the root it is colored red. Possibly "badness" (red node having red parent) is moving towards the root.

**Case 3:** (terminates insertion). One operation of Treenode and recoloring is sufficient, see the figure.

Case 2

Case 3
Red-Black trees: insertion, example

Sequence of insertions:

1. Insert 2:
   - Tree: 1 → 1
   - Insert: 2
   - Result: 1 → 1 → 2

2. Insert 3:
   - Tree: 1 → 1 → 2
   - Insert: 3
   - Result: 1 → 1 → 2 → 3

3. Case 1:
   - Tree: 1 → 1 → 2 → 3
   - Insert: 1
   - Result: 1 → 1 → 2 → 3 → 1

4. Insert 4:
   - Tree: 1 → 1 → 2 → 3 → 1
   - Insert: 4
   - Result: 1 → 1 → 2 → 3 → 1 → 4

5. Case 2:
   - Tree: 1 → 1 → 2 → 3 → 1 → 4
   - Insert: 2
   - Result: 1 → 1 → 2 → 3 → 1 → 2

6. Insert 5:
   - Tree: 1 → 1 → 2 → 3 → 1 → 2
   - Insert: 5
   - Result: 1 → 1 → 2 → 3 → 1 → 2 → 5

7. Case 1:
   - Tree: 1 → 1 → 2 → 3 → 1 → 2 → 5
   - Insert: 4
   - Result: 1 → 1 → 2 → 3 → 1 → 2 → 5 → 4

8. Insert 6:
   - Tree: 1 → 1 → 2 → 3 → 1 → 2 → 5 → 4
   - Insert: 6
   - Result: 1 → 1 → 2 → 3 → 1 → 2 → 5 → 4 → 6

9. Case 2:
   - Tree: 1 → 1 → 2 → 3 → 1 → 2 → 5 → 4 → 6
   - Insert: 3
   - Result: 1 → 1 → 2 → 3 → 1 → 2 → 5 → 4 → 6 → 3
**Red-Black trees: insertion, example cont.**

Sequence of insertions:

Case 1

Case 2

Case 3: \( x=6 \), black uncle=3

make Treenode(2,4,6)
Red-Black trees: deletion

The deletion of any node is reduced to the deletion of a leaf. The deletion of a red leaf is a single action, no restructuring is needed. Several cases for the deletion of the black leaf are illustrated below. In some cases we have to create a node which is a double-black node.

Deletion of a double-black.

Four cases are considered. Assume that \( v \) is the double-black node, \( y \) is its brother, and \( z \) is its father. The sons of \( y \) are "nephews" of the double-black node. The crucial point is that if the double-black is the root the we can change it to single black directly.

Case 1: If \( y, z \) are black, both nephews are black then make \( v \) single-black and make \( z \) double-black. \( v := z \). The double-black moves toward the root. Iterate this case until (not Case 1 or \( v = root \)).

Case 2: If \( y \) is black, both nephews are black and \( z \) is red make the recoloring: color(y)=red; color(z)=black, This terminates the deletion.

Case 3: If \( y \) is black and a nephew \( x \) is red we do Treenode\( (x, y, z) \) and make suitable recoloring. This terminates the deletion.

Case 4: Otherwise \( y \) is red and both nephews are black. If \( v \) is a right sone choose \( x \) to be the left son of \( y \), otherwise \( z \) is the right son. Perform Treenode\( (x, y, z) \) and make suitable recoloring. Afterwards we have Case 2 or Case 3.
There are four cases, in Case 1 double-black goes up, eventually it disappears, the double-black root can be changed to black. None of the cases 2, 3, 4 leads to Case 1.
The problem "Towers of Hanoi"

Towers of Hanoi

H(n,x,y,z) move n disks from tower x to y using tower z

H(n,x,y,z) {if (n>0) { H(n−1,x,z,y); move n from tower x to y; H(n−1,z,y,z); } }

The number of moves is T(n) = if (n=1) then 0 else 2 T(n−1) + 1

T(n) = 2^n − 1
Iterative versions of the "Towers of Hanoi".

Algorithm:
   do
     if $n$ is odd then direction is anticlockwise
     otherwise it is clockwise;
     each second move, starting from the first move
     move the element 1 according to the direction;
     In other moves move another element
   while the final situation is not reached.

How to generate the sequence of elements which are being moved. Create the sequence of
empty-set slots.

   $\emptyset \emptyset \emptyset \emptyset \emptyset \ldots$

Place 1 in every second empty-set slot starting with the first empty-set slot.
We obtain the sequence;

   $1 \emptyset 1 \emptyset 1 \emptyset \emptyset \ldots$

Now place 2 in every second empty-set slot.
We get:

   $1 2 1 \emptyset 1 2 1 \emptyset \ldots$

Then we insert 3 in every second empty slot etc.
Consequently we get the sequence:

   $1 2 2 3 1 2 1 4 1 2 2 3 1 2 1 \ldots$

If $n$ is odd then we know that element 1 is all the time moving to its anticlockwise neighboring
tower, element 2 travels clockwise, element 3 anticlockwise etc.
In this way we can easily compute on which tower is each element without simulating the whole
game.
The sequence can be also generated by a recurrence:

   $S_1 = 1; \quad S_n = S_{n-1} \cdot n \cdot S_{n-1}$
A fast method to compute the whole configuration after $m$ moves.

(optional part)

We assume here that the towers are numbered 0, 1, 2 and we move elements from tower 0 to tower 1. We construct an auxiliary sequence $x_1, x_2, \ldots$ of ternary sequences, the sequence depends on the parity of $n$.

<table>
<thead>
<tr>
<th>n is even</th>
<th>n is odd</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 = 00000001$</td>
<td>00000002</td>
</tr>
<tr>
<td>$x_2 = 000000021$</td>
<td>000000012</td>
</tr>
<tr>
<td>$x_3 = 000000122$</td>
<td>000000211</td>
</tr>
<tr>
<td>$x_4 = 000021111$</td>
<td>000012222</td>
</tr>
<tr>
<td>$x_5 = 00012222$</td>
<td>00021111</td>
</tr>
<tr>
<td>$x_6 = 00211111$</td>
<td>00122222</td>
</tr>
</tbody>
</table>

We represent $m$ in binary as $[m]_2$, and count bits from the right, the first bit has number 1. We "add" the numbers $x_i$ corresponding to positions in $[m]_2$ which contain 1. The positions in $[m]_2$ are counted from the right starting with 1. The addition is modulo 3 without carry. The resulting sequence $y = \ldots d_n d_{n-1} \ldots d_2 d_1$ shows on which tower is which element. We read the last $n$ ternary numbers from $y$. The $i$-th element is on the tower with index $d_i$.

For example if $n = 7$ and $m = 13$ we have $[13]_2 = 1101$. Then we do addition mod 3 without carry of $x_4$, $x_3$, $x_1$ since 1101 contains 1 on positions 4,3,1 (counting from the right). Then we do addition in columns mod 3.

\[
\begin{array}{c}
0000002 \\
0000211 \\
0001222 \\
\hline
0001102 \\
\end{array}
\]

We read on which towers are the disks by counting from the right. This means that element 1 is on tower 2, elements 3, 4 are on the tower 1, and all other elements (2,5,6,7) on the tower 0.
Another application of binary representations

The game Nim

Nim is a game played by two parties. There are three numbers \(a, b, c\). In each move the player reduces one of numbers by a positive integer. The players play alternatively. The winner is the player making the last move.

Nim is a very simple game. It is based on Boolean arithmetic and, in addition, a binary representation of integers.

The most important operation for Nim is exclusive or or XOR denoted by \(^\).

XOR applies bitwise to the binary representation of two or more numbers.

For example, if \(x=7\) and \(y=11\) are two decimal numbers then their binary representations will be

\[(x)2=111 \text{ and } (y)2=1011, \text{ respectively.}\]

Therefore, \(x ^ y = 1100\).

Then the winning move should leave the situation with

\[a ^ b ^ c = 0.\]

Hence the winning positions are such that

\[a ^ b ^ c \text{ is not } 0\]
Huffman coding, application of priority queues

Priority queue can be applied to solve Huffman coding problem in \( O(n \cdot \log n) \) time.

Assume we have a text \( x \) consisting of \( n \) letters with repetitions. the letter \( i \) appears \( w_i \) times in \( x \). We need to encode the letters in binary in a way to minimize the total length of encoding and guarantee that it is prefix free. None of the code is a prefix of another one.

The codes can be represented as a binary tree, with left edges corresponding to zero, and right edge corresponding to one. Let \( S = \{w_1, w_2, \ldots, w_n\} \). Denote by \( \text{HuffmanCost}(S) \) the total cost of minimal encoding and by \( \text{HuffmanTree}(S) \) the tree representing an optimal encoding. Observe that several different optimal trees are possible.

We have:

\[
\text{HuffmanCost}(S) = \text{HuffmanCost}(S - \{u, w\}) \cup \{u + w\} + (u + w),
\]

where \( u, w \) are two minimal elements of \( S \).

This implies the following algorithm, assume that initially \( S \) is stored in a min-priority queue.

\begin{verbatim}
HuffmanCost(S)
{
  if S contains only one element $u$ then return $u$;

  u = Extract Min(S); w = ExtractMin(S);

  insert(u+w,S);

  return HuffmanCost(S)+u+w
}
\end{verbatim}

The algorithm can be easily written without recursion. The complexity is \( O(n \log n) \) since each operation in the priority queue takes \( O(\log n) \) time.

**Remark.** If the weights are given in a sorted order then optimal Huffman code can be constructed in linear time.
There is possible an algorithm using ”normal” queue with constant time operations of inserting and deleting element from the queue.

Algorithm 2
{
initialize empty queue Q; total_cost = 0;

sort initial n elements into nondecreasing order and

push them on the stack S, smallest elements closer to the top;

while |Q|+|S| >2 {

    let u, w be the smallest elements chose from the first two
    element of Q and the first top elements of S;

    remove u, w from Q and S;

    insert u+w into the queue Q;

    total_cost += u+w;}

return total_cost
}
More recursive algorithms: permutation generation

Recursive generation of all permutations of 1 ...n.

```c++
#include <swap.h>

template<class T>
inline void Swap(T& a, T& b)
{// Swap a and b.
    T temp = a; a = b; b = temp;
}

// output all permutations of n elements
#include "swap.h"

template<class T>
void Perm(T list[], int k, int m)
{// Generate all permutations of list[k:m].
    int i;
    if (k == m) {// list[k:m] has one permutation, output it
        for (i = 0; i <= m; i++)
            cout << list[i];
        cout << endl;
    }
    else {// list[k:m] has more than one permutation
        // generate these recursively
        for (i = k; i <= m; i++) {
            Swap(list[k], list[i]);
            Perm(list, k+1, m);
            Swap(list[k], list[i]);
        }
    }
}

void main(void)
{
    char a[] = {'1', '2', '3', '4'};
    int n = 3;
    cout << "The permutations of 123 are" << endl;
    Perm(a, 0, n-1);
}
```
**Permutation generation: nonrecursive**

Non-recursive generation of all permutations.

**Algorithm 1:**

generate permutations by transpositions of adjacent elements.

Active element, max. element with arrow pointing to a smaller adjacent neighbor. Exchange it with the pointed neighbor. Change arrows for larger elements.

Start in the situation as in the example. Finish when there is no move.

\[
\begin{array}{ccc}
\leftarrow 1 & \leftarrow 2 & \leftarrow 3 \\
\rightarrow 3 & \leftarrow 2 & \leftarrow 1 \\
\leftarrow 1 & \leftarrow 3 & \leftarrow 2 \\
\rightarrow 3 & \leftarrow 1 & \leftarrow 2 \\
\end{array}
\]

**Algorithm 2:**

generate permutations in lexicographic order.

Start with 1 2 3 ... n. Each time find maximal element \( k \) (assume it is on some position \( i \)) whose right neighbor is larger. Exchange it with the smallest element to the right which is larger than \( k \) and reverse the suffix of the last \( i - 1 \) elements.

Start with 1,2, ..., n. Finish when there is no move.

\[
\begin{array}{cccccccc}
1234 & 1243 & 1324 & 1342 & 1423 & 1432 \\
2134 & 2143 & 2314 & 2341 & 2413 & 2431 & \text{etc.}
\end{array}
\]
permutation generation: another recursive algorithm

```
Heap’s-Generation(n)
{
    if n=1 output A[1 .. n]
    else
        for i=1 to n do
            { Heap’s-Generation(n-1);
              if odd(n) swap A[1] with A[n]
              else
                  swap A[i] with A[n] }
}
```
Graphs

A graph $G = (V, E)$ consists of a set $V$ of vertices (nodes) and a set $E$ of edges (connections between vertices). There are directed and undirected graphs.

Two basic representations of a graph: (1) Adjacency matrix. (2) Adjacency lists.

**DFS (DepthFirstSearch)**

Initially all nodes are unmarked, visit the node $v$, then recursively visit each subgraph starting from neighbours of $v$. In this process visited vertices are marked and not processed again. As a result we have a *dfs-tree* which gives some useful structure of a graph.

```plaintext
void DFS(v) {if v is unmarked
    {visit v; mark v;
     for each vertex w adjacent to v DFS(w); }}
```

**BFS (BreadthFirstSearch or Level-order traversal)**

```plaintext
void BFS(v) {
    initialize a queue to contain v;

    while queue is nonempty
        {v := dequeue(queue);
         if v is unmarked
             {visit v; mark v; for each vertex w adjacent to v enqueue(w);
             } }

    void main()

    { Initialize each vertex to be unmarked;

     for each vertex v BFS(v); }
```

If the graph is undirected both BFS and DFS compute *connected components*. The graph is strongly connected *iff there is a directed path between any pair of vertices.*
DFS and BFS trees

Adjacency lists representation of G

1:  2, 4, 8
2:  1, 6
3:  4, 5
4:  1, 3, 5
5:  3, 4
6:  2, 9, 8
7:  10
8:  1, 6
9:  6
10: 4, 7

DFS spanning tree

BFS tree
Applications of DFS

DFS gives a simple algorithm to test connectivity of the undirected graph. The situation with directed graph is more complicated.

Denote by $\tilde{G}$ the reversed version of a directed graph $G$, each directed edge is reversed.

Algorithm Testing strong-connectivity of the whole digraph;

test using DFS if all vertices are reachable in $G$ from vertex 1;
test using DFS if all vertices are reachable in $\tilde{G}$ from vertex 1;
if both tests are positive then return true; else return false

Topological sorting of acyclic graphs.

If the graph is directed and acyclic (no cycles) then topological sorting is any permutation of vertices $v_1, v_2, \ldots, v_n$ such that $i < j$ implies there is no edge from $j$ to $i$ (possibly there is an edge from $i$ to $j$, but not necessarily).

The algorithm DFS can be used to sort topologically directed graph in the following way:

In the function DFS(v) add at the end one extra statement:

insert v at the beginning of list L;

Algorithm TopologicalSorting1

list L = empty list;

for i=1 to n do

if i is not visited then DFS(i);

permute the vertices in the order they are on L;
The indegree (outdegree) of a vertex is the number edges incoming to (outgoing from) the vertex.
There is possible a different algorithm for topological sorting:

Algorithm TopologicalSorting2

list L = empty list;

for i=1 to n do

{ find a vertex v in V with outdegree 0;
  insert v at the beginning of L;
  remove v and all its incoming edges; }

We can keep the vertices with outdegree zero in a queue (or a stack, does not matter), also we keep table OUTDEGREE[] of outdegrees, each time a vertex is removed outdegree of its neighbours decreases by one, each time outdegree of a vertex becomes zero we insert in into the queue. Then finding a vertex with outdegree zero is in constant time, and updating table OUTDEGREE[] takes constant time per each removed edge. Altogether we have a simple linear time algorithm.

Strongly connected components.
A subset of vertices is a strongly connected component if each vertex is reachable by a directed path from each other in the subset, and it is an inclusion-maximal subset with this property.

The algorithm for strongly connected components is quite tricky. We use DFS to compute the list L as in TopologicalSorting1, even if the graph is not acyclic. Then we apply DFS to the reversed graph using the reversed list L, as starting nodes for DFS. Each time a single application of DFS gives a new strongly connected component.

Assume visit(v) in DFS means: component(v) = counter.

Algorithm Strongly connected Components (not done in class)

  reverse list L; all vertices are initially unvisited; counter=1;
  for each v in L in the order given by L do
    { DFS(v); counter++ }
Strongly connected components of an example graph (not done in class):
Strong orientation of an undirected graph, all DFS tree edges are oriented from the root, other edges towards the root. It works if there is no bridge (an edge which after removing disconnects the graph).
Greedy Algorithms
Dijkstra’s algorithm for single-source shortest path

Single-source shortest path problem Given a graph \( G = (V, E) \) with \( d_{ij} > 0 \) the weight of an edge \((i, j)\) (if no such edge exists, \(d_{ij} = \infty\)), and a source vertex \(s\), find the cost of every minimal total weight path between \(s\) and any other vertex of \(G\). Edge weights MUST BE POSITIVE for the algorithm to work properly. NOTE: Remember that shortest refers to total edge weight NOT number of edges of the path!

Dijkstra(G,d,s)
1. for every \(v\) in \(V\) do
2. if \((s,v)\) is an edge \(D(v)=d(s,v);//D(v)\) to hold shortest path weight from \(s\) to \(v\)
3. \(S=\{s\}\);
4. \(Q=V-S=V-\{s\}\);
5. while \(Q\) is not empty do
6. \(\text{EXTRACT from } Q \text{ vertex } u \text{ with } \text{MINIMUM } D(u) \) //EXTRACT-MINIMUM
7. \(S=S \cup \{u\}\);
8. for each \(v\) in \(Q\) such that \((u,v)\) is an edge do
9. \(\text{update } D(v) = \min(D(v), D(u)+d(u,v))\)

The above algorithm is what we call a greedy algorithm. A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. For a split \(S\) and \(V - S\) in Dijkstra’s algorithm, that vertex of \(V - S\) is added into \(S\) that is closest (lightest) to any node of \(S\).

Greedy algorithms do not yield in general optimal results. This is however the case with Dijkstra’s Algorithm. Spanning tree problems is another class of problems that can be solved by using greedy algorithms.
Dijkstra’s Algorithm
Correctness and Running time

Proof of correctness (by induction).

Inductive hypothesis.
1. For \( u \in S \), \( D(u) \) is the length of the shortest path from \( s \) to \( u \).
2. For \( u \notin S \), \( D(u) \) is the shortest path from \( s \) to \( u \) with intermediate vertices in \( S \) other than \( u \) of course.

Base case In the beginning \( S = \{ s \} \). Therefore the minimum \( D(u) \) gives the weight of the lightest edge connecting \( s \) directly to a vertex in \( V - S \). This vertex is \( u \), which is added to \( S \) and (1) becomes true. All neighbors \( v \) of \( u \) then update their \( D(v) \); the shortest path which is internal to \( S \) from \( s \) to \( v \) is (a) either a single edge path from \( s \) to \( v \) (i.e. an edge) of cost \( D[v] = d(u,v) \) (Step 2 of the algorithm), or a path from \( s \) that goes through \( u \); in that case \( D[v] \) is updated, if necessary, by the length of that path, i.e. \( D[u] + d(u,v) \).

Inductive Step. Let the inductive hypothesis apply to any set \( S \) of size \( k \); i.e. \( |S| = k \). We are going to prove that the hypothesis is true for a set \( S \) such that \( |S| = k + 1 \) (inductive step). Let \( u \) be the vertex outside \( S \) with minimum \( D(u) \). Then the shortest path from \( s \) to \( u \) is internal in \( S \) except for a single edge that goes from a vertex of \( S \) directly to \( u \). For otherwise (i.e. the path was not internal in \( S \)) there would be another vertex in that path that from \( s \) to \( u \) that is also in \( V - S \) and that is closer to \( s \) than \( u \), a contradiction to the minimality of \( D(u) \) (NOTE that this is the case because positive weight edges are only allowed).

Running time of the algorithm. Suppose that \( Q \) uses an adjacency list representation of \( G \). Line 6 is performed \( |V| - 1 \) and finding the minimum takes time \( O(|V|) \) each iteration for a total time for line 5 of \( O(|V|^2) \). A vertex extracted from \( Q \) gets into \( S \) once and only once. Whenever this happens all its neighbors are examined. Therefore lines 8-9 are executed once for every edge for a total of \( O(|E|) \). Total running time is \( O(|E| + |V|^2) \).

If we implement \( Q \) with a priority queue, lines 1-2 build a queue \( Q \) in \( O(|V|) \) time. Line 6 is performed \( |V| - 1 \) times for the cost of Extract-Min from a heap in \( O(\lg |V|) \) steps for a total cost of \( O(|V|\lg |V|) \). Lines 8-9 are executed once for every edge. Each time a Decrease-Key operation is performed for a total cost of \( O(|E|\lg |V|) \). Therefore total time is \( O(|E| + |V|^2) \). Note this running time is better for sparse graphs than \( O(|E| + |V|^2) \).
Greedy Algorithms for minimum spanning trees

Introduction

A spanning tree of an undirected connected graph $G$ is a tree that saturates/touches every vertex of the graph.

Fact. Every spanning tree of a graph with $n$ vertices has $n - 1$ edges.

If graph edges have weights, a minimum spanning tree is a spanning tree for which the sum of its weights is minimum. We present below two algorithms for finding minimum spanning trees. Both algorithms are greedy.

Kruskal’s algorithm is described below.

Kruskal’s Algorithm(G,V,E)
1. T= {}; // empty set
2. while (T has fewer than n-1 edges)
3. add to T the shortest edge that does not make T have a cycle

Prim’s algorithm is described below.

Prim’s Algorithm(G,V,E)
1. T=empty; X={r}; Q= V: // r is an arbitrary vertex of V
2. while ( X != V ) {
3. let e=(u,v) be lowest cost edge from u in X to v in Q
4. T= T U {(u,v)};
5. X=X U {v}; Q= Q - {v};
6. }

If $G = (V,E)$ is a connected undirected graph and $S = (V,T)$ is a spanning tree of $G$, then

1. For all $v_1, v_2 \in V$ a path between $v_1, v_2$ is unique.
2. Any edge in $E - T$ added to $S$ creates a unique cycle.
3. If an edge in $E - T$ is added to $T$ and then another edge is removed from the resulting cycle, we get a new spanning tree.
Greedy Algorithms for spanning trees
Prim’s Algorithm

Prim(G,V,E,W) // W: weight matrix i.e. weight of edge (u,v) is w(u,v)
1. Q=V;
2. foreach v in V do {
3. cost[v]= infinity;
4. p(v)  = NIL ; //parent of v initialization
5. }
6. cost[r] = 0; p(r) = NIL;  // Set X= {r} in previous pseudocode
7. while Q is not empty {
8. u=EXTRACT-MIN(Q);
9. for each v in Adj(u) and v in Q {
10. if w(u,v) < cost[v] {
11. p(v)=u;
12. cost[v]=w(u,v);
13. }
14. }
15. }
**Proposition Prim.** Let \( X \subset V \). If \((u,v)\) is the lowest cost edge from \( X \) to \( Q = V - X \), then there is a minimum cost spanning tree that includes \((u,v)\).

**Proof (by contradiction).**
1. Let us assume by way of contradiction that all minimum spanning trees DO NOT contain \((u,v)\).
2. Let \( T' \) be a minimum spanning tree and by 1. it does not include \((u,v)\).
3. Consider edge \((u,v)\). Since \( T' \) is a spanning tree there is a path in that tree that connects \( u \) to \( v \). This path does not include course edge \((u,v)\) by way of 1.
4. Consider \( T' \cup \{(u,v)\} \). Subgraph has a cycle. The remainder applies to this subgraph of \( G \).
5. In the path from \( u \in X \) to \( v \in V - X \) there must exists another vertex \( u' \in X \), and \( v' \in V - X \) such that \((u',v') \in E\).
6. Prim's algorithm chose \((u,v)\) over \((u',v')\), i.e. \( w(u,v) \leq w(u',v') \).
7. Consider \( N = T' - (u',v') + \{(u,v)\} \).
8. \( N \) has \((u,v)\) and \( \text{cost}(N) = \text{cost}(T') - w(u',v') + w(u,v) \leq \text{cost}(T') \)
9. \( N \) is a minimum spanning tree that includes \((u,v)\), which is a contradiction to 1.

**Running Time.** We implement \( Q \) with a binary heap. Steps 1-5 take \( O(|V|) \) time. Each vertex is extracted exactly once and the number of vertices checked in line 8 is at most \( 2|E| \) ie \( O(|E|) \). Line 7 takes \( O(|V| \lg |V|) \) time, lines 9-11 are implemented using a Decrease-Key operation that requires \( O(\lg |V|) \) time per call for a total of \( O(|E| \lg |V|) \). Total time is therefore \( O((|E|+|V|) \lg |V|) \). Note that the test in line 9 of membership in \( Q \) is implemented by having one extra bit per vertex. This bit is set to 1 for a vertex in \( Q \). For a removed vertex it is set to 0.
Greedy Algorithms for minimum spanning trees

Kruskal’s Algorithm

Kruskal(G,V,E,w)
1. T=empty;
2. for each v in V
3. do Make-Set(u);
4. sort edges of E by nondecreasing weight w;
5. for each edge (u,v) in E in nondecreasing weight order
6. do if Find-Set(u) != Find-Set(v)
7. then T=T U {(u,v)} ;
8. Union(u,v);

Proof of correctness(by contradiction). Graph T = {e_1, \ldots, e_{n-1}} is definitely a tree that spans G if G is connected. Suppose that T is not a minimal spanning tree. Let T' be the minimal spanning tree of G with the maximum number of edges common with T. Suppose e_k = (a, b) be the first edge in T not in T'. Let P be the path in T' that connects a, b (as edge (a, b) is not in T'). Let G_k be the connected component containing \{e_1, \ldots, e_{k-1}\} that saturates a. Then there exists an edge e in P connecting G_k to a vertex outside G_k. Since the algorithm chooses e_k over e in T \ w(e_k) \leq w(e). But in T' - e + e_k is a spanning tree at least as cheap as T', but with one more edge in common with T than T' a contradiction to the choice of T'.

Running Time. Steps 1-3 take O(|V|) time. Sorting takes O(|E|\lg|E|) time. Steps 5, 6, 7 are executed O(|E|) times if an adjacency list representation of the graph is considered, and total cost by prior discussion is O(|E|\lg|E|) as well.
Floyd-Warshall’s algorithm

**Algorithm** FloydWarshall(G,A,d)
//no negative weight cycles
// d(i,j) one-edge distances, output - shortest path distances
1. \( D^0(i, j) = d(i, j) \)
2. for k=1 to n
   for i=1 to n
      for j=1 to n
         \( D^k(i, j) = \min(D^{k-1}(i, j), D^{k-1}(i, k) + D^{k-1}(k, j)) \);
3. OUTPUT the matrix \( D^n \);

**Proof of correctness (by Induction).**

**Inductive assumption.** After loop \( k = l \) is executed, \( D^l_{ij} \) is the cost of the minimal weight path from \( i \) to \( j \) that ONLY USES internal vertices from \( \{1, \ldots, l\} \). (NOTE that \( i,j \) can be larger than \( l \)).

**Basis of induction.** Before the loop is executed for the first time, all paths with no internal vertices are single edges; \( D^0_{ij} = d_{ij} \) is the cost of an edge path between \( i \) and \( j \) with NO internal vertices. Subsequently, let us assume that the **Inductive assumption** is true for \( l \). We shall prove it for \( l+1 \).

Case 1. A shortest path \( P \) from \( i \) to \( j \) using internal vertices from \( \{1, \ldots, l, l+1\} \) goes from \( i \) into \( l+1 \) at most once (with internal vertices from \( \{1, \ldots, l\} \)) and then from \( l+1 \) goes to \( j \) (with internal vertices from \( \{1, \ldots, l\} \)). It cannot go through \( l+1 \) more than once because this would indicate the existence of a cycle. If such a cycle was part of \( P \) as all edge weights are positive (or if negative weights are allowed, there are no negative weight cycles), this cycle could be removed the resulting path \( P' \) from \( i \) to \( j \) has total weight less than that of \( P \), a contradiction to the minimality of \( P \). In \( P \), \( D^l_{i,l+1} \) is the cost of the first subpath of \( P \), and \( D^l_{i+1,j} \) is the cost of the second one, and line 5 would detect such a path \( P \) if it exists. Therefore \( D^{l+1}_{ij} \) will be correctly updated.

Case 2. A shortest path \( P \) from \( i \) to \( j \) with intermediate vertices in \( \{1, \ldots, l, l+1\} \) never goes through \( l+1 \). Then line 5 of the \( k = l + 1 \) loop does nothing.

**Note.** Floyd’s algorithm finds ALL minimal weight paths between any two vertices. Its running time is \( O(|V|^3) \). It works even with negative edge weights as long as there are no negative weight cycles.