Jerzy Nawrocki
Faculty of Computing & Telecommunications
Poznan University of Technology
jerzy.nawrocki@put.poznan.pl

Computational Complexity

https://suewhite.com.au/wp-content/uploads/2013/05/Timer-stopwatch-347x300.jpg/

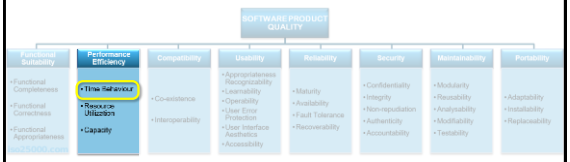
Tentative schedule of lectures

No.	Topic	Date
1	Imperative Programming	2023-10-09
2	Digital Circuits	2023-10-16
3	Computers	2023-10-23
4	Subprograms	2023-11-06
5	Text Processing	2023-11-13
6	Object-oriented Programming	2023-11-20
7	Numerical methods	2023-11-27
8	Computational Complexity	2023-12-04
9	Databases and Machine Learning	2023-12-11
10	Parallel Processing	2023-12-18
11	Computer Networks & Cybersecurit	2024-01-08
12	Software Engineering	2024-01-15
13	Embedded Systems	2024-01-22
14	Professionalism in Computing	2024-01-29

Scope of the individual test

No.	Topic	Date
1	Imperative Programming	2023-10-09
2	Digital Circuits	2023-10-16
3	Computers	2023-10-23
4	Subprograms	2023-11-06
5	Text Processing	2023-11-13
6	Object-oriented Programming	2023-11-20
7	Numerical methods	2023-11-27
8	Computational Complexity	2023-12-04
9	Databases and Machine Learning	2023-12-11
10	Parallel Processing	2023-12-18
11	Computer Networks & Cybersecurit	2024-01-08
12	Software Engineering	2024-01-15
13	Embedded Systems	2024-01-22
14	Professionalism in Computing	2024-01-29

ISO/IEC 25010: Software quality model



https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Aim of the lecture

- How to analyze the **speed of an algorithm** with mathematical rigor?
- What are the **limits of computing?**

Agenda

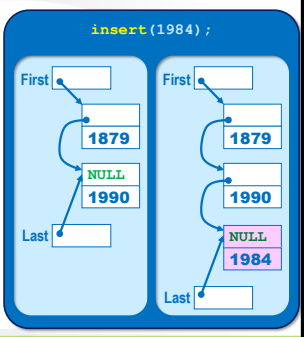
- **Priority queue and heap**
- **Heapsort and selection sort**
- **Big O notation**
- **Sorting by brute force**
- **Polynomial-time reduction**
- **The halting problem**

Queues

- **FIFO queue**
- LIFO queue
- Priority queue

FIFO queue: First In First Out

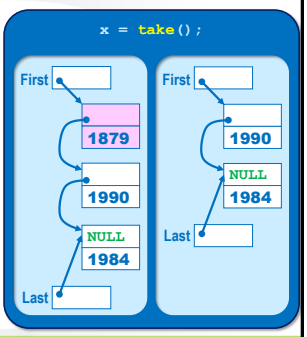
```
insert(1984);
```



- `insert(e)`
- `take()`

FIFO queue: First In First Out

```
x = take();
```



- `insert(e)`
- `take()`

FIFO queue: First In First Out

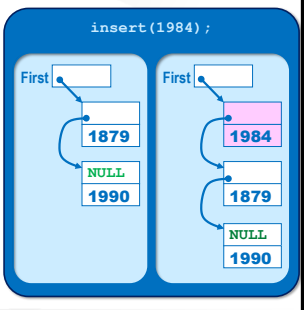
- `insert(e)`: `append(e)`
- `take()`: `pop(0)`

Queues

- FIFO queue
- **LIFO queue**
- Priority queue

LIFO queue: Last In First Out

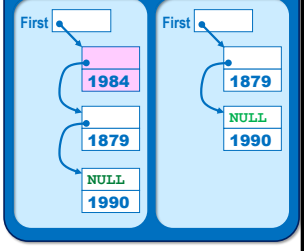
```
insert(1984);
```



- `insert(e)`
- `take()`

LIFO queue: Last In First Out

```
x = take()
```

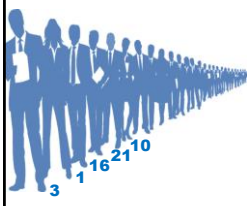


- `insert(e)`
- `take()`

LIFO queue: Last In First Out

- `insert(e)`: `insert(0, e)`
- `take()`: `pop(0)`

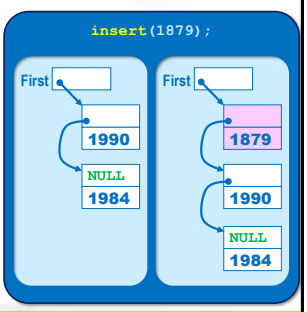
Queues



- FIFO queue
- LIFO queue
- **Priority queue**

Priority queue

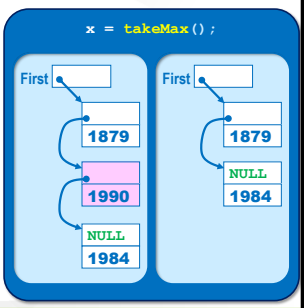

```
insert(1879);
```



- `insert(e)`
- `takeMax()`

Priority queue


```
x = takeMax();
```



- `insert(e)`
- `takeMax()`

Priority queue
can be implemented by means of
heap.

Undirected graph

G = <V, E>

Undirected graph

G = <V, E>
E ⊂ {{a,b}: a ∈ V, b∈V, a≠b }

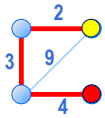
Path

G = <V, E>
E ⊂ {{a,b}: a ∈ V, b∈V, a≠b }

Path

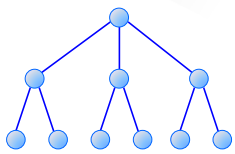
G = <V, E>
E ⊂ {{a,b}: a ∈ V, b∈V, a≠b }

Shortest Path Problem



G = <V, E>
E ⊂ {{a,b}: a ∈ V, b∈V, a≠b }

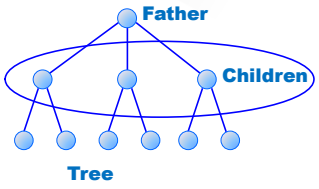
Tree



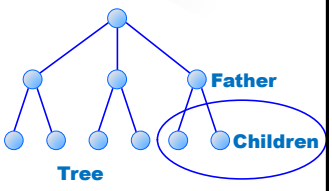
Tree

There is one path between any two vertices.

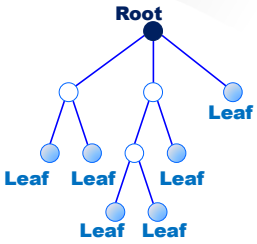
Tree



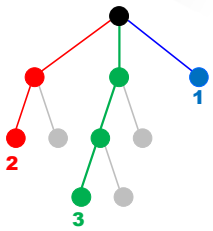
Tree



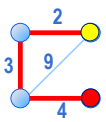
Tree



Path length



Shortest Path Problem

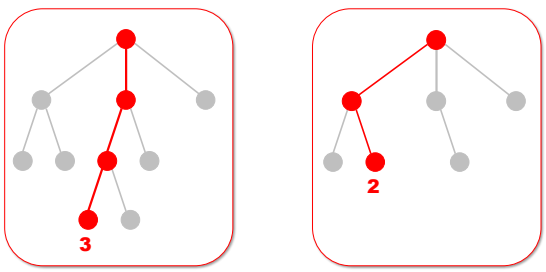


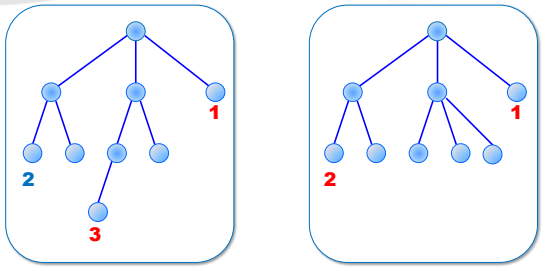
$G = \langle V, E \rangle$

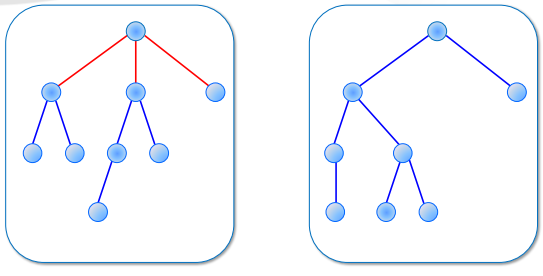
$E \subset \{\{a,b\}: a \in V, b \in V, a \neq b\}$

Height of a tree

Height of a tree = length of the longest path



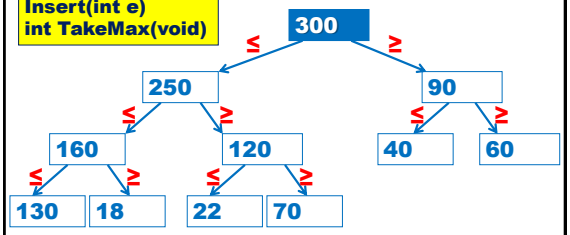




Priority queue as a max-heap

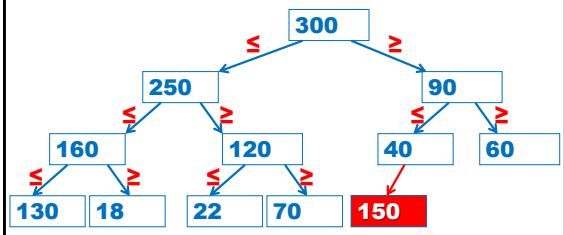
Value (Father) ≥ Value (Son)

Insert(int e)
int TakeMax(void)



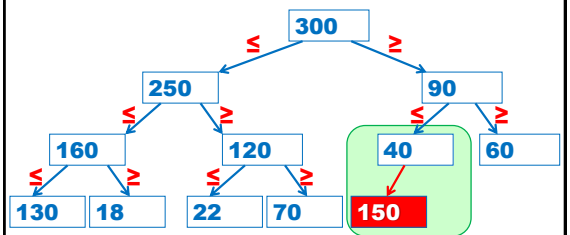
Priority queue as a max-heap

Value (Father) ≥ Value (Son)



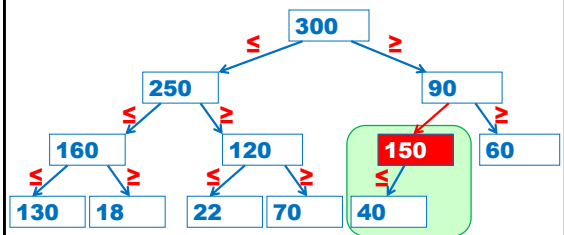
Priority queue as a max-heap

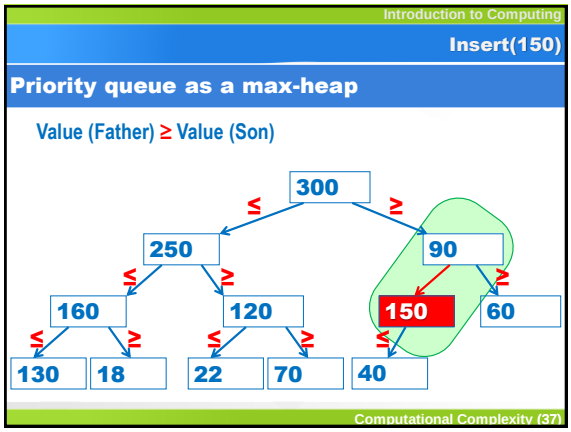
Value (Father) ≥ Value (Son)

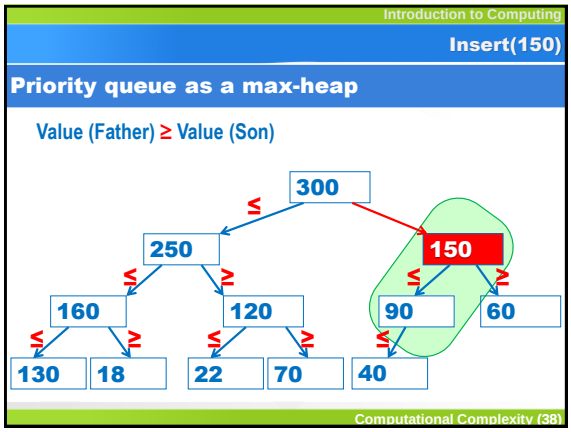


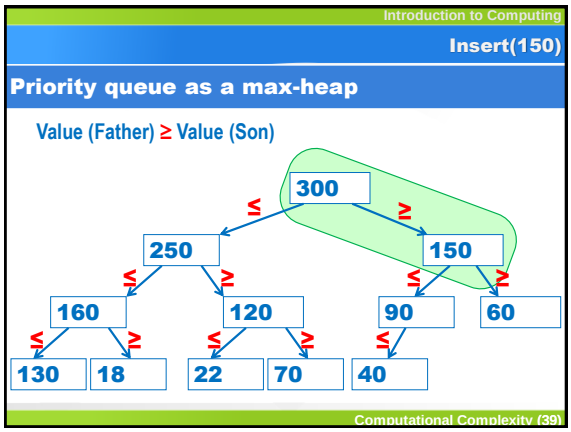
Priority queue as a max-heap

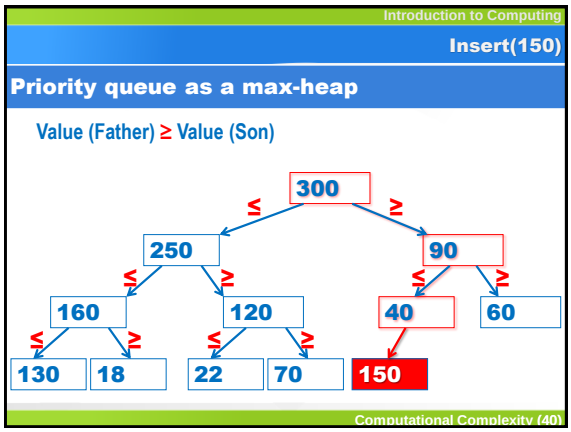
Value (Father) ≥ Value (Son)

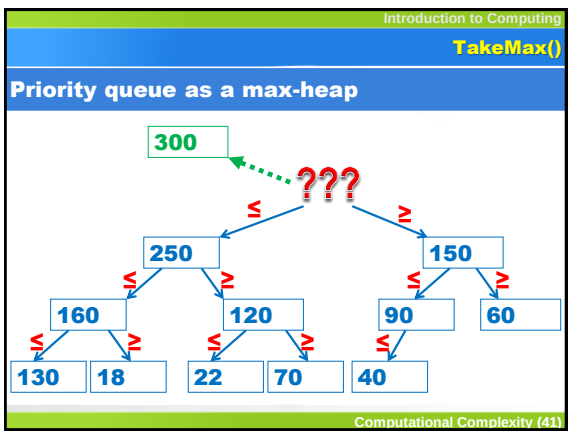


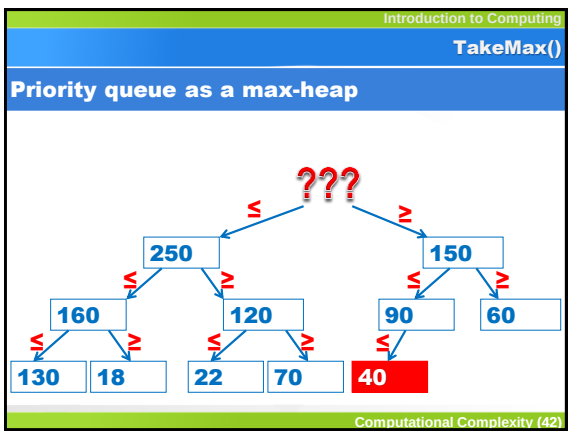

Introduction to Computing
Insert(150)
Priority queue as a max-heap
Value (Father) ≥ Value (Son)
300
250
90
160
120
150
60
130
18
22
70
40
Computational Complexity (37)

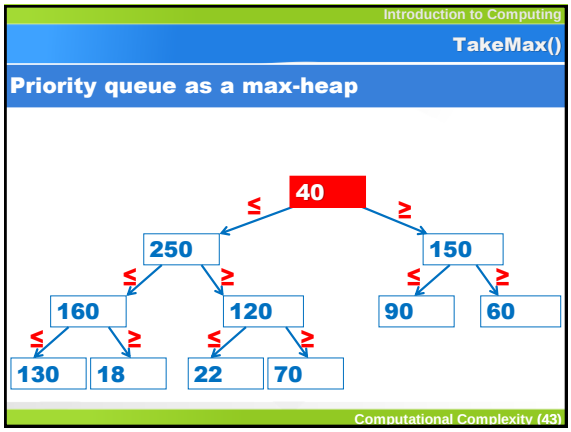

Introduction to Computing
Insert(150)
Priority queue as a max-heap
Value (Father) ≥ Value (Son)
300
250
150
160
120
90
60
130
18
22
70
40
Computational Complexity (38)

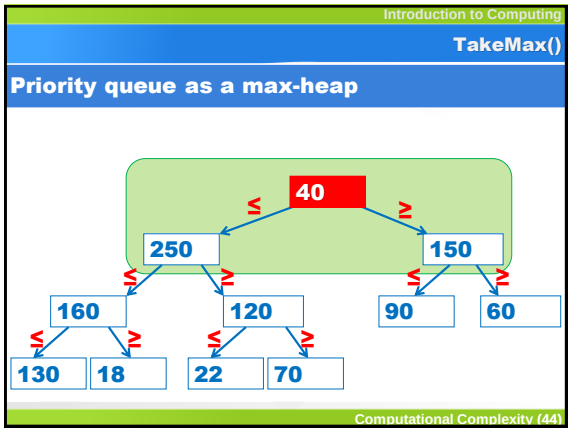

Introduction to Computing
Insert(150)
Priority queue as a max-heap
Value (Father) ≥ Value (Son)
300
250
150
160
120
90
60
130
18
22
70
40
Computational Complexity (39)

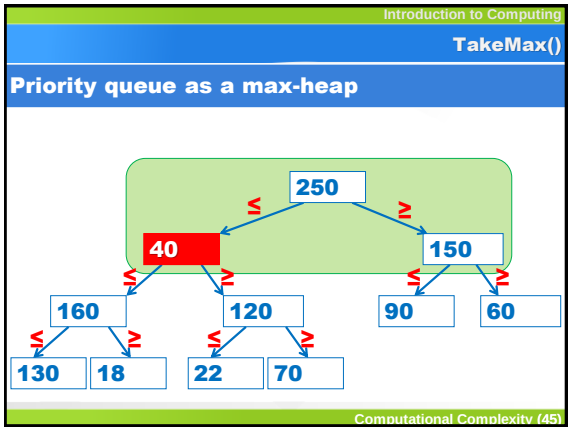

Introduction to Computing
Insert(150)
Priority queue as a max-heap
Value (Father) ≥ Value (Son)
300
250
90
160
120
40
60
130
18
22
70
150
Computational Complexity (40)

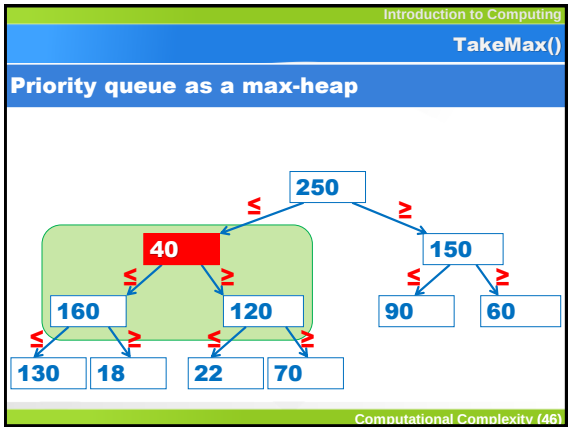

Introduction to Computing
TakeMax()
Priority queue as a max-heap
300
???
250
150
160
120
90
60
130
18
22
70
40
Computational Complexity (41)

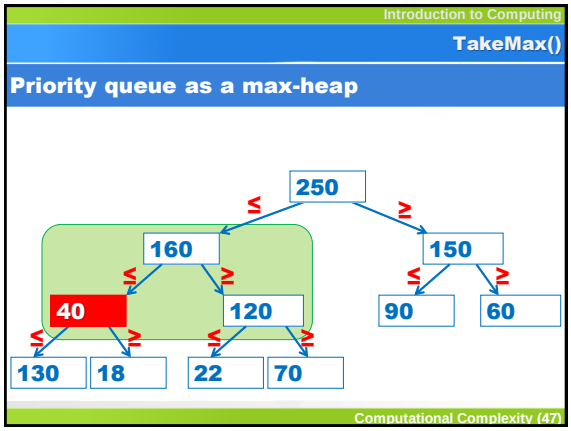

Introduction to Computing
TakeMax()
Priority queue as a max-heap
???
250
150
160
120
90
60
130
18
22
70
40
Computational Complexity (42)

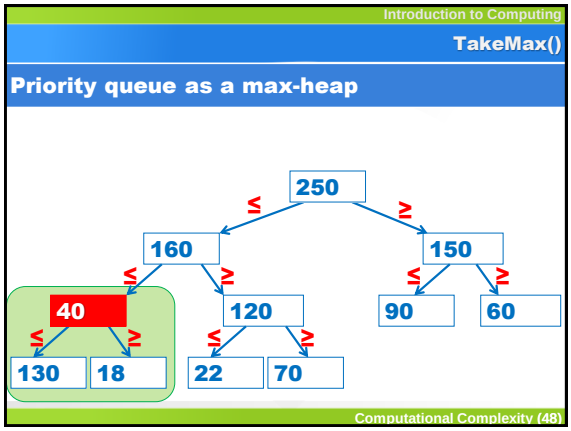

Introduction to Computing
TakeMax()
Priority queue as a max-heap
40
250
150
160
120
90
60
130
18
22
70
Computational Complexity (43)


Introduction to Computing
TakeMax()
Priority queue as a max-heap
40
250
150
160
120
90
60
130
18
22
70
Computational Complexity (44)


Introduction to Computing
TakeMax()
Priority queue as a max-heap
250
40
150
160
120
90
60
130
18
22
70
Computational Complexity (45)

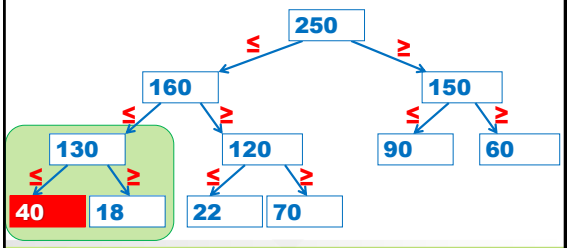

Introduction to Computing
TakeMax()
Priority queue as a max-heap
250
40
150
160
120
90
60
130
18
22
70
Computational Complexity (46)


Introduction to Computing
TakeMax()
Priority queue as a max-heap
250
160
150
40
120
90
60
130
18
22
70
Computational Complexity (47)


Introduction to Computing
TakeMax()
Priority queue as a max-heap
250
160
150
40
120
90
60
130
18
22
70
Computational Complexity (48)

TakeMax()

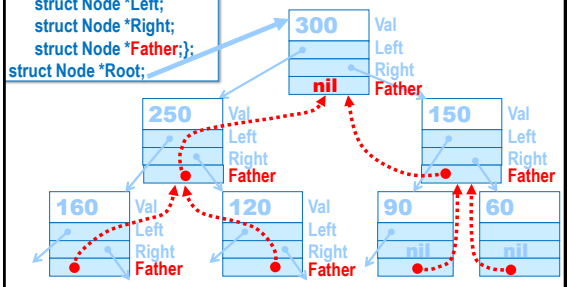
Priority queue as a max-heap



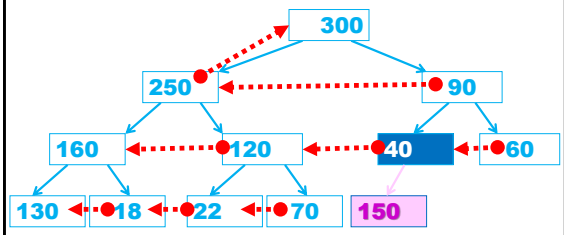
Heap implementation

Pointer-based implementation

```
struct Node {int Val;
    struct Node *Left;
    struct Node *Right;
    struct Node *Father;};
struct Node *Root;
```

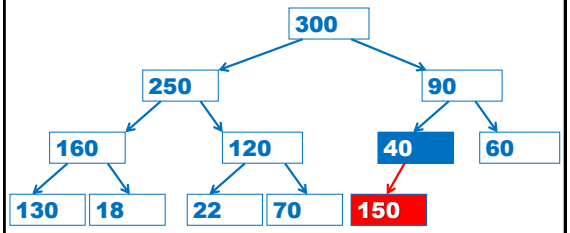


Finding the father-to-be node



Dynamic storage allocation?

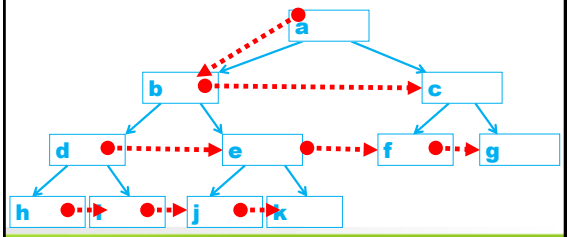
```
malloc(..);
```



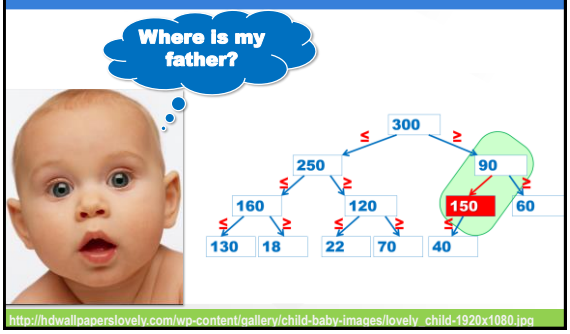
First-fit strategy

Pointer-less binary tree

a	b	c	d	e	f	g	h	i	j	k
1	2	3	4	5	6	7	8	9	10	11



Priority queue as a max-heap

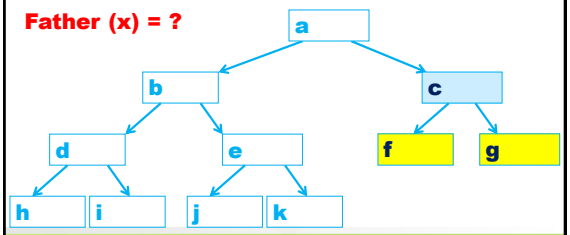


http://hdwallpaperslovely.com/wp-content/gallery/child-baby-images/lovely_child-1920x1080.jpg

Pointer-less binary tree

a	b	c	d	e	f	g	h	i	j	k
1	2	3	4	5	6	7	8	9	10	11

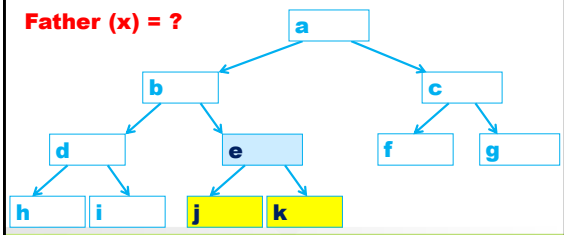
Father (x) = ?



Pointer-less binary tree

a	b	c	d	e	f	g	h	i	j	k
1	2	3	4	5	6	7	8	9	10	11

Father (x) = ?



Pointer-less binary tree

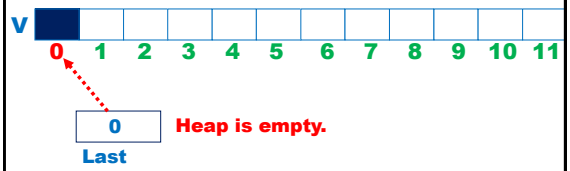
a	b	c	d	e	f	g	h	i	j	k
1	2	3	4	5	6	7	8	9	10	11

Father (x)= entier(x/2)

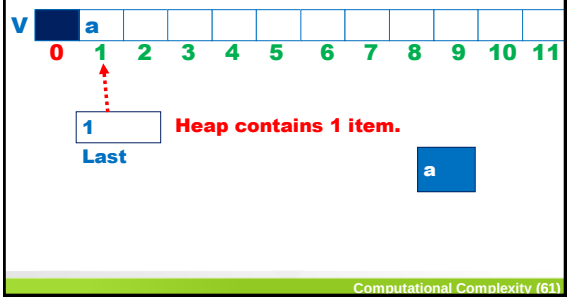
Father (6) = 3
Father (7)= 3
Father (10)= 5
Father (11)= 5

C	Python
(int)x/2	x // 2

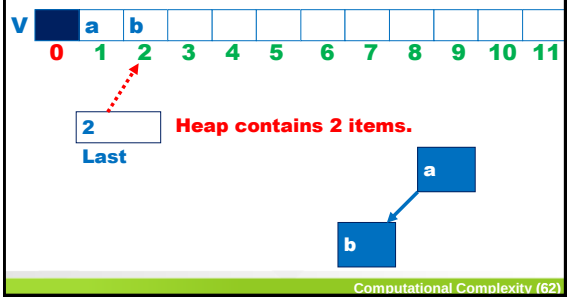
Pointer-less binary tree



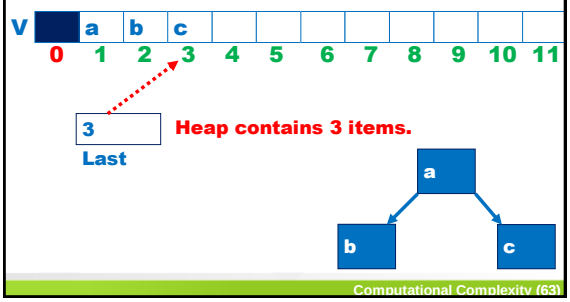
Pointer-less binary tree



Pointer-less binary tree



Pointer-less binary tree



Agenda

- Priority queue and heap
- **Heapsort and selection sort**
- Big O notation
- Sorting by brute force
- Polynomial-time reduction
- The halting problem

Heapsort

1 16 2 2 10

16 10 2 2 1

1) insert, insert, ... insert
2) takeMax, takeMax, ..., takeMax

Write a C program that reads integers X_1, .., X_n and displays the sequence X_i sorted from max to min.

Heapsort

```
. . .
#include <stdio.h>
int main(void){
    int x;
    while (scanf("%d", &x) != EOF){
            insert(x);
            }
    while (!empty()){
            printf("%d  ", takeMax());
            }
    printf("\n");
    return 0; }
```

Selection sort

```c
#define  MaxSize 100
#define  NONE -1
#include <stdio.h>
 int S[MaxSize], Last=-1;
 int IndexMax(void){
     int i, Max;
     Max= 0;
     for(i=1; i<=Last; i++)
         if(S[Max] < S[i]) Max= i;
     return Max;
     }
 int main(void){
     int i,Max, Indx;
     for(; scanf("%d",&S[++Last]) != EOF;)
         ;
     for(i=1; i <= Last; i++){
         Max= S[Indx=IndexMax()];
         S[Indx]= NONE;
         printf("%d,  ", Max);
         }
     return 0;}
```

Selection sort

```c
#define  MaxSize 100
#define  NONE -1
#include <stdio.h>
 int S[MaxSize], Last=-1;
 int IndexMax(void){
     int i, Max;
     Max= 0;
     for(i=1; i<=Last; i++)
         if(S[Max] < S[i]) Max= i;
     return Max;
     }
 int main(void){
     int i,Max, Indx;
     for(; scanf("%d",&S[++Last]) != EOF;)
         ;
     for(i=1; i <= Last; i++){
         Max= S[Indx=IndexMax()];
         S[Indx]= NONE;
         printf("%d,  ", Max);
         }
     return 0;}
```

Selection sort

```c
#define  MaxSize 100
#define  NONE -1
#include <stdio.h>
 int S[MaxSize], Last=-1;
 int IndexMax(void){
     int i, Max;
     Max= 0;
     for(i=1; i<=Last; i++)
         if(S[Max] < S[i]) Max= i;
     return Max;
     }
 int main(void){
     int i,Max, Indx;
     for(; scanf("%d",&S[++Last]) != EOF;)
         ;
     for(i=1; i <= Last; i++){
         Max= S[Indx=IndexMax()];
         S[Indx]= NONE;
         printf("%d,  ", Max);
         }
     return 0;}
```

Introduction to Computing

Agenda

- Priority queue and heap
- Heapsort and selection sort
- **Big O notation**
- Sorting by brute force
- Polynomial-time reduction
- The halting problem

Computational Complexity (70)

Introduction to Computing

Sorting

1 16 2 2 10

16 10 2 2 1

Heapsort | **Selection sort** | . . . | **Algorithm *n***

Which of them is the fastest one?

Computational Complexity (71)

Introduction to Computing

Execution speed – How to describe it?

Execution time

It depends on:
- **the amount of data**
- **type of computer**

Computational Complexity (72)

Sorting – Impact of the amount of data

1 16 2 2 10
16 10 2 2 1

Ladies	Gentlemen
1 16 2	1 16 2 35 12 7 21 15 14 33 1 17 20 23 15
n = 3	n = 15

Execution speed – How to describe it?

Execution time

It **depends on:**

- **the amount of data**
- **type of computer** (type of microprocessor, cache memory etc.)

Execution speed – How to describe it?

Number of fixed-time steps

~~Execution time~~

It **depends on:**

- **the amount of data**
- ~~**type of computer**~~ (type of microprocessor, cache memory etc.)

Write a C program that reads integers x_1, .., x_n and displays the sequence x_i in the reverse order.

```
#include <stdio.h>
#define MAX  100
int main(void){
    int x, Top= MAX, Stack[MAX];
    while (scanf("%d", &x) != EOF){
           Stack[--Top]= x;
           }
    while (Top < MAX){
           printf("%d  ", Stack[Top++]);
           }
    printf("\n");
    return 0;
    }
```

How many steps?

2n

2n +1

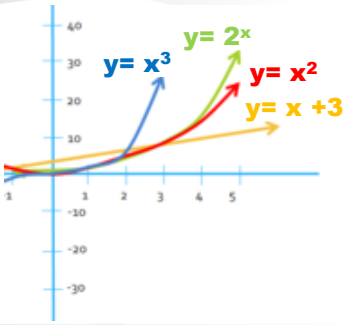
n

Is this distinction really important?

When the size of input data is big ...

IMPORTANT: Any quadratic function grows faster than linear one, cubic faster than quadratic etc.

UNIMPORTANT: Constant coefficients



How to treat this with mathematical rigor?

O(n)

A non-negative function f(x) is of order of another non-negative function g(x)

f(x) is O(g(x))

iff there exists a constant C such that for almost all* non-negative values of x the following inequality holds:

$f(x) \le C \cdot g(x)$.

* Almost all = all except a finite number

O(n)

A non-negative function f(x) is of order of another non-negative function g(x)

f(x) is O(g(x))

iff there exists a constant C such that for almost all non-negative values of x (i.e. all except a finite number) the following inequality holds:

$f(x) \le C \cdot g(x)$.

In other words, **when neglecting constant coefficients**

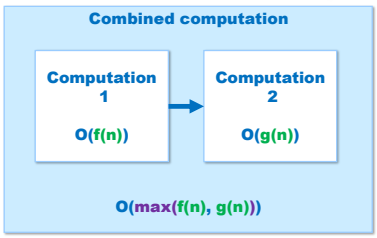
f(x) is growing not faster than g(x).

Examples

f(x)=2x is O(x)

$f(x)=2x^2 + 3x +1$ is $O(x^2)$

f(x)= 15 is O(1)



Write a C program that reads integers $x_1, .., x_n$ and displays the sequence x_i in the reverse order.

```
#include <stdio.h>
#define MAX  100
int main(void){
    int x, Top= MAX, Stack[MAX];
    while (scanf("%d", &x) != EOF){
           Stack[--Top]= x;
           }
    while (Top < MAX){
           printf("%d  ", Stack[Top++]);
           }
    printf("\n");
    return 0;
    }
```

How many steps?

2n

2n +1

n

Write a C program that reads integers $x_1, .., x_n$ and displays the sequence x_i in the reverse order.

```
#include <stdio.h>
#define MAX  100
int main(void){
    int x, Top= MAX, Stack[MAX];
    while (scanf("%d", &x) != EOF){
           Stack[--Top]= x;
           }
    while (Top < MAX){
           printf("%d  ", Stack[Top++]);
           }
    printf("\n");
    return 0;
    }
```

How many steps?

O(n)

O(n)

Write a C program that reads integers x_1, .., x_n and displays the sequence x_i in the reverse order.

```
#include <stdio.h>
#define MAX  100
int main(void){
    int x, Top= MAX, Stack[MAX];
    while (scanf("%d", &x) != EOF){
           Stack[--Top]= x;
           }
    while (Top < MAX){
           printf("%d  ", Stack[Top++]);
           }
    printf("\n");
    return 0;
    }
```

How many steps?

O(n)

Puzzle

What is the time complexity of the push function?

```
#define MAX 100
int Top, Stack[MAX];

void push (int e){
        Top= Top - 1;
        Stack[Top] = e;
        return;
        }
```

O(1)

In this lecture

Worst / Average **case**

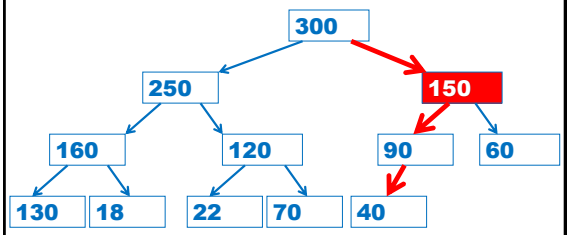
- time complexity — *execution time*
- space complexity — *memory consumption*

Question

Worst-case time complexity of **heapsort**?

Worst-case time complexity of **insert()**?

Insert(150)



Question

Worst-case time complexity of heapsort?

Worst-case time complexity of **insert()**?

How the **hight of the tree**, **h**, depends on the **number of nodes**, **n**?

#Nodes	Height
2 .. 3	1
4 .. 7	2
8 .. 15	3
. . .	. . .
2^h .. $(2^{h+1} - 1)$	h

n = #Nodes

$n = 2^h \cdot \varepsilon$

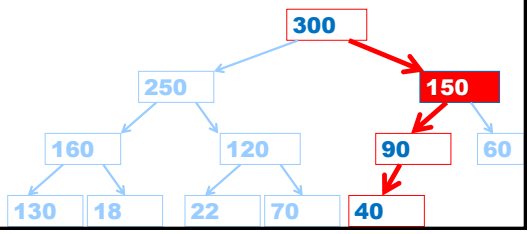
$1 \le \varepsilon < 2$

$n/\varepsilon = 2^h$

$\log_2 n/\varepsilon = h$

$h = \log_2 n - \log_2 \varepsilon$

$h = O(\log_2 n)$



Write a C program that reads integers X_1, .., X_n and displays the sequence X_i sorted from max to min.

Heapsort

```
. . .
#include <stdio.h>
int main(void){
    int x;
    while (scanf("%d", &x) != EOF){
           insert(x);
           }
    while (!empty()){
           printf("%d  ", takeMax());
           }
    printf("\n");
    return 0; }
```

O(log n)

n O(log n) → O(n log n)

Question

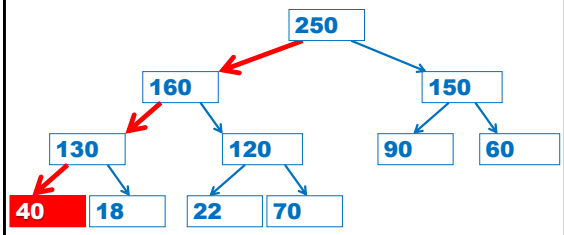
Worst-case time complexity of heapsort?

Worst-case time complexity of insert()?

Worst-case time complexity of **takeMax**?

TakeMax()

Priority queue as a max-heap



Write a C program that reads integers X_1, .., X_n and displays the sequence X_i sorted from max to min.

Heapsort

```
. . .
#include <stdio.h>
int main(void){
    int x;
    while (scanf("%d", &x) != EOF){
           insert(x);
           }
    while (!empty()){
           printf("%d  ", takeMax());
           }
    printf("\n");
    return 0; }
```

O(1)

O(log n)

n O(log n) → O(n log n)

n O(log n) → O(n log n)

Write a C program that reads integers X_1, .., X_n and displays the sequence X_i sorted from max to min.

Heapsort

```
. . .
#include <stdio.h>
int main(void){
    int x;
    while (scanf("%d", &x) != EOF){
           insert(x);
           }
    while (!empty()){
           printf("%d  ", takeMax());
           }
    printf("\n");
    return 0; }
```

n O(log n) → O(n log n)

n O(log n) → O(n log n)

TOTAL: O(n log n)

Question

Worst-case time complexity of heapsort?

Worst-case time complexity of insert()?

Worst-case time complexity of takeMax?

O(n log n)

Write a C program that reads integers X_1, .., X_n and displays the sequence X_i in the reverse order.

```
. . .
#include <stdio.h>
int main(void){
    int x;
    while (scanf("%d", &x) != EOF){
            push(x);
            }
    while (!empty()){
            printf("%d  ", pop());
            }
    printf("\n");
    return 0; }
```

Homework: What is worst-case complexity of the main function?

```
typedef int Bool;
#define MAX 100
 int Top= MAX, Stack[MAX];
void push (int e){
     Stack[--Top]= e;
     return;
     }
 int pop (){
     return Stack[Top++];
     }
Bool empty (){
     return Top == Max;
     }

 int main(void){
     . . .
     }
```


Agenda

- Priority queue and heap
- Heapsort and selection sort
- Big O notation
- **Sorting by brute force**
- Polynomial-time reduction
- The halting problem

Brute force approach

Quality checker

Possible solution

Solution generator

(generates all possible solutions one by one)

Brute force – Sorting

Quality checker

Checks if the order is preserved

Possible solution (data)

Solution generator

Returns permutations, one by one

Brute force sorting with the SearchPerm function

```
def SearchPerm(s, Cond, Act):
    # Cond = permutation selection condition
    # Act = action on the selected permutation
    # Can be used for sorting, scheduling etc.
# ----- The case of sorting: -----
def Ascend(s):
    for i in range(1, len(s)):
        if s[i-1] > s[i]:
            return False
    return True
Seq= [1, 55555, 333, 4444]
SearchPerm(Seq, Ascend, print)
```

```
[1, 333, 4444, 55555]
```


Implementation of SearchPerm – Iterative version

```
def SearchPerm(s, Cond, Act):
    perm= Init(s)
    while perm != []:
        if Cond(perm):
            Act(perm)
            return
        perm= GetNext()
```

Possible solution

```
def Init(L):
    . . .
    return CurList
```

Possible solution

```
def GetNext():
    . . .
    return CurList
```


Implementation of SearchPerm – Iterative version

```
def Init(L):
    global Config
    global IniList
    global CurList
    IniList= L
    Config= []
    CurList= []
    for i in range(len(IniList)):
        Config.append(0)
        CurList.insert(0, IniList[i])
    return CurList
```


Implementation of SearchPerm – Iterative version

```
def GetNext():
    global IniList
    global Config
    global CurList
    j= len(IniList) - 1
    while j > 0 and Config[j] == j:
        CurList.pop(j)
        Config[j]= 0
        j-= 1
    if  j > 0:
        CurList.pop(Config[j])
        Config[j]+= 1
        CurList.insert(Config[j], IniList[j])
    else:
        return []
    for i in range(j+1, len(IniList)):
        Config[i]= 0
        CurList.insert(0, IniList[i])
    return CurList
```


Implementation of SearchPerm – Recursive version

```
def Permut(IniList, Lev, CurList, Act, Cond):
    # IniList = sequence to be permuted
    # Act = Action on the correct sequence
    # Cond = Condition defining correct sequence
    E= IniList[Lev]
    for p in range(len(CurList)+1):
        CurList.insert(p, E)
        if Lev == len(IniList) - 1:
            if Cond(CurList):
                Act(CurList)
        else:
            Permut(IniList, Lev+1, CurList, Act, Cond)
        CurList.pop(p)
def SearchPerm(s, Cond, Act):
    Permut(s, 0, [], Act, Cond)
```


Computational intractability

Permutation of a set = a sequence of all its elements (each one appears only once)

Permutations of {A}: A

Permutations of {A, B}: BA, AB

Permutations of {A, B, C}: CBA, BCA, BAC
CAB, ACB, ABC

C(1) = 1 ns

Implementation of SearchPerm – Iterative version

```
def SearchPerm(s, Cond, Act):
    perm= Init(s)
    while perm != []:
        if Cond(perm):
            Act(perm)
            return
        perm= GetNext()
```

Possible solution

```
def Init(L):
    . . .
    return CurList
```

Possible solution

```
def GetNext():
    . . .
    return CurList
```


Computational intractability

Permutation of a set = a sequence of all its elements (each one appears only once)

Permutations of {A}: A

Permutations of {A, B}: BA, AB

Permutations of {A, B, C}: CBA, BCA, BAC
CAB, ACB, ABC

C(1) = 1 ns
C(2) = 2 • C(1) = 2 ns
C(3) = 3 • C(2) = 6 ns
C(n) = n • C(n-1) = n! • C(1)
. . .
C(26) = ...?

C(26) ≈ 12 788 288 741 years

Agenda

- Priority queue and heap
- Heapsort and selection sort
- Big O notation
- Sorting by brute force
- **Polynomial-time reduction**
- The halting problem

Knapsack problem

Items:
- **weight w_i,**
- **value v_i**

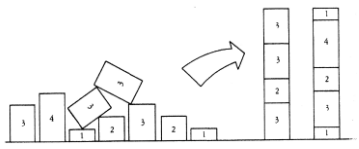
Knapsack:
- **capacity C**

Optimization problem: What is the maximum total value of items one can put into the knapsack?

Decision problem: Can we put subset of items of the total value not less than K?

https://en.wikipedia.org/wiki/File:Knapsack_Problem_Illustration.svg

Partition problem



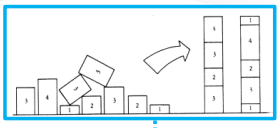
Each item has a size s_i.

Decision problem: Can the set be split into 2 subsets of the same total size?

https://codegolf.stackexchange.com/questions/48486/the-partition-problem-sorting-stacks-of-boxes

Problem reduction

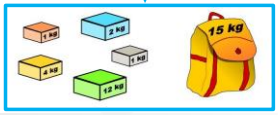
No polynomial-time algorithm



Polynomial-time

$w_i = s_i$
$v_i = s_i$
$C = 0.5 \Sigma s_i$
$K = 0.5 \Sigma s_i$

?



Agenda

- Priority queue and heap
- Heapsort and selection sort
- Big O notation
- Sorting by brute force
- Polynomial-time reduction
- **The halting problem**

Puzzle

```
a, b = 21, 28
while b > 0:
    a, b = b, a % b
print(a)
```

7

```
prog= '''
a, b = 21, 28
while b > 0:
    a, b = b, a % b
print(a)
'''
p= compile(prog, "", "exec")
eval(p)
```

7

Another puzzle

```
prog= '''
a, b = 21, 28
while b > 0:
    a, b = b, a # <-- change
print(a)
'''
p= compile(prog, "", "exec")
eval(p)
```

Time limit exceeded #stdin #stdout 5s

Code analyser – Halting problem

```
prog= '''
a, b = 21, 28
while b > 0:
    a, b = b, a # <-- change
print(a)
'''
def WillHalt(p):
    # some magic code comes here
    if p == prog:
        return False
```

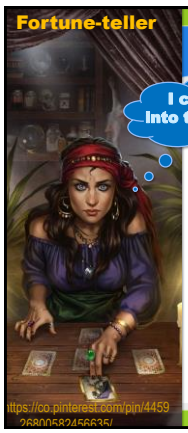

Halting problem

```
prog= '''
a, b = 21, 28
while b > 0:
    a, b = b, a % b
print(a)
'''
def WillHalt(p):
    # some magic code comes here
    if p == prog:
        return True

p= compile(prog, "", "exec")
if WillHalt(prog):
    eval(p)
    print("It halts as predicted.")
else:
    eval(p)
    print("Surprise! It has halted!")
```

Is it possible to write this function?

Auxiliary problem



How to prove she's wrong?

Halting problem is undecidable

```
prog= '''
while WillHalt(prog):
      pass
print(13)
'''
def WillHalt(p):
    # some magic code comes here
```


Halting problem

```
prog= '''
while WillHalt(prog):
      pass
print(13)
'''
def WillHalt(p):
    # some magic code comes here
    if p == prog:
       return False
p= compile(prog, "", "exec")
if WillHalt(prog):
   eval(p)
   print("It halts as predicted.")
else:
   eval(p)
   print("Surprise! It has halted!")
```

```
13
Surprise - it has halted!
```


Halting problem

```
prog= '''
while WillHalt(prog):
      pass
print(13)
'''
def WillHalt(p):
    # some magic code comes here
    if p == prog:
       return True  # <--- changed opinion ----
p= compile(prog, "", "exec")
if WillHalt(prog):
   eval(p)
   print("It halts as predicted.")
else:
   eval(p)
   print("Surprise! It has halted!")
```

Time limit exceeded #stdin #stdout 5s 9724KB

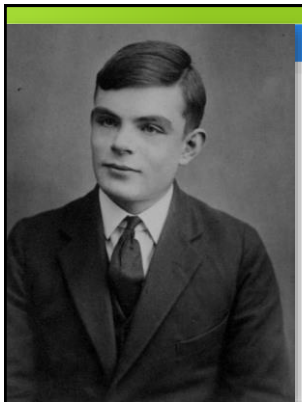
Halting problem

```
prog= '''
while WillHalt(prog):
      pass
print(13)
'''

def WillHalt(p):
```

The halting problem is undecidable.

Alan Turing (1912-1954)



The halting problem is undecidable.

On Computable Numbers, with an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, 1936

https://en.wikipedia.org/wiki/Alan_Turing

Universal Turing Machine

S_0	0	0	R	S_0	1	1	R	S_0	...	Δ	1	0	1	*b*

Program | Data

	0	1	*b*
S_0	<0, R, S_0>	<1, R, S_0>	<*b*, L, S_1>
S_1	<0, R, Parz>	<1, R, Niep>	-

Halting problem

```
prog= '''
while WillHalt(prog):
      pass
print(13)
'''

def WillHalt(p):
```

The halting problem is undecidable.

We are not able to analyse every program in an automatic way.

Fermat's Last Theorem

$a^n + b^n = c^n$ for $n > 2$

Pierre de Fermat

https://en.wikipedia.org/wiki/Pierre_de_Fermat

Fermat's Last Theorem

$a^n + b^n = c^n$ for $n > 2$

Sir Andrew Wiles

```
p= 2
def Fermat():
    a= 2
    while True:
       b= 2
       while b <= a:
          sum= a**p + b**p
          if int((sum)**(1/p))**p == sum:
             return (a, b)
          else:
             b+= 1
       a+= 1
print(Fermat())
```

```
(4, 3)
```

https://en.wikipedia.org/wiki/Andrew_Wiles

Fermat's Last Theorem

$a^n + b^n = c^n$ for $n > 2$

Sir Andrew Wiles

```
p= 3
def Fermat():
    a= 2
    while True:
       b= 2
       while b <= a:
          sum= a**p + b**p
          if int((sum)**(1/p))**p == sum:
             return (a, b)
          else:
             b+= 1
       a+= 1
print(Fermat())
```

If the halting problem was decidable ...

https://en.wikipedia.org/wiki/Andrew_Wiles

Agenda

Summary

Agenda

- **Priority queue and heap**
- **Heapsort and selection sort**
- **Big O notation**
- **Sorting by brute force**
- **Polynomial-time reduction**
- **The halting problem**