Axioms

- Time for fetching - τ_{fetch}
- Time for storing - τ_{store}

  Example: `y = x` - time is $\tau_{fetch} + \tau_{store}$
- Time for elementary operations $+$, -, $\times$, $\div$, $<$, etc, are denoted by $\tau_{+}, \ldots, \tau_{<}$.
- Time for call and return τ_{call} and τ_{return} are constant.
- Time for passing an argument to a method is same as τ_{store}.

  Example: `x = f(x)` - time is $\tau_{fetch} + 2 \times \tau_{store} + \tau_{call} + \tau_{return} + \tau_{f(x)}$

Example: Compute $\sum_{1}^{n} i$

The algorithm is

```
1: result = 0;
2: for (int i=1; i<=n; i++)
3:    result += i;
4: return result;
```

line No.	Time
1	$\tau_{fetch} + \tau_{store}$
2a	$\tau_{fetch} + \tau_{store}$
2b	$(n+1) \times (2 \times \tau_{fetch} + \tau_{<})$
2c	$n \times (2 \times \tau_{fetch} + \tau_{+} + \tau_{store})$
3	$n \times (2 \times \tau_{fetch} + \tau_{+} + \tau_{store})$
4	$\tau_{fetch} + \tau_{return}$
total:	$(6\tau_{fetch} + 2\tau_{store} + \tau_{<} + 2\tau_{+}) \times n +$ $(5\tau_{fetch} + 2\tau_{store} + \tau_{<} + \tau_{return})$

Array subscripting operation

- `y = a[i]` has the running time =

$$3\tau_{fetch} + \tau_{[.]} + \tau_{store}$$

- Three fetches: `a`, `i` and `a[i]`
- Time for address calculation of `a[i]` is $\tau_{[.]}$.

Example

```
result = a[n];
for (int i=n-1; i>=0; --i)
   result = result*x+a[i];
return result;
```

Example contd.

line No.	Time
1	$3\tau_{fetch} + \tau_{[.]} + \tau store$
2(i)	$2\tau_{fetch} + \tau_{-} + \tau store$
2(ii)	$(n+1) \times (2\tau_{fetch} + \tau_{<})$
2(iii)	$n \times (2\tau_{fetch} + \tau_{-} + \tau store)$
3	$n \times (5\tau_{fetch} + \tau_{[.]} + \tau_{*} + \tau_{+} + \tau store)$
4	$\tau_{fetch} + \tau_{return}$
total	$n(9\tau_{fetch} + 2\tau_{store} + \tau_{<} + \tau_{[.]} + \tau_{+} + \tau_{*} + \tau_{-}) + (8\tau_{fetch} + 2\tau_{store} + \tau_{[.]} + \tau_{-} + \tau_{<} + \tau_{return})$

Recursive functions

- $n! = \begin{cases} 1, \ n = 0, \\ n(n-1)! \ n > 0, \end{cases}$

```
if (n==0)
   return 1;
else
   return n*factorial(n-1);
```

- Running time

line	$n = 0$	$n > 0$
1	$2\tau_{fetch} + \tau_{<}$	$2\tau_{fetch} + \tau_{<}$
2	$\tau_{fetch} + \tau_{return}$	-
4	-	$3\tau_{fetch} + \tau_{-} + \tau_{store} + \tau_{\times} +$
		$\tau_{call} + \tau_{return} + T(n-1)$

- Therefore, $T(n) = T(n-1) + t_2$, where $t_2 = 5\tau_{fetch} + \tau_{<} + \tau_{-} + \tau_{store} + \tau_{\times} + \tau_{call} + \tau_{return}$

Another example: find max

```
result=a[0];
for(i=1; i<n; i++)
   if (a[i] > result)
      result=a[i];
return result;
```

line No.	Time
1	$3\tau_{fetch} + \tau_{[.]} + \tau_{store}$
2a	$\tau_{fetch} + \tau_{store}$
2b	$n(2\tau_{fetch} + \tau_{<})$
2c	$(n-1)(2\tau_{fetch} + \tau_{+} + \tau_{store})$
3	$(n-1)(4\tau_{fetch} + \tau_{[.]} + \tau_{<})$
4	$(3\tau_{fetch} + \tau_{[.]} + \tau_{store}) \times\ ?$
5	$\tau_{fetch} + \tau_{return}$

Analysis of running time

- Note line 4 will be executed if

$$a_i > \left(\max_{0 \le j < i} a_j\right)$$

- Implying

$$T(n) = t_1 + nt_2 + \sum_{i=1,\ a_i > \max a_j}^{n-1} t_3$$

where

$$\begin{aligned} t_1 &= 2\tau_{store} - \tau_{fetch} - \tau_{+} - \tau_{<} \\ t_2 &= 8\tau_{fetch} + 2\tau_{<} + \tau_{[.]} + \tau_{+} + \tau_{store} \\ t_3 &= 3\tau_{fetch} + \tau_{[.]} + \tau_{store} \end{aligned}$$

Average running time

- Average running time means that if the algorithm is executed large number of times on a selection of random sets of size n each - what is the average time ?
- Depends on input distribution
  - descending sorted sequence - line 4 is executed always
  - ascending sorted sequence - line 4 is never executed

Average running time

- But no a priori knowledge of the input can be assumed. Thus, line 4 is executed with a probability p_i, where

$$p_i = P\left[a_i > \left(\max_{0\le j<i} a_j\right)\right]$$

- Line 4 is executed only if a_i is largest of $i+1$ values $a_0, a_1, \ldots, a_i$. Probability of this event is $\frac{1}{i+1}$

$$P\left[a_i > \left(\max_{0\le j<i} a_j\right)\right] = \frac{1}{i+1}$$

- Therefore, average running time is

$$\begin{aligned} T_{average}(n) &= t_1 + nt_2 + \textstyle\sum_{i=1}^{n-1} p_i t_3 \\ &= t_1 + nt_2 + \textstyle\sum_{i=1}^{n-1} \frac{1}{i+1} t_3 \\ &= t_1 + nt_2 + t_3(\textstyle\sum_{i=1}^{n-1} \frac{1}{i} - 1) \\ &= t_1 + nt_2 + t_3(H_n - 1) \end{aligned}$$

where H_n is the nth harmonic number.

Harmonic series

- The series is non convergent $H_n - H_{n-1} = \frac{1}{n}$
- But H_n represents the so called descending staircase function

$$\int_1^n \frac{1}{\lfloor x \rfloor} = H_{n-1}$$

- So an approximation could be the area under the smooth curve

$$\int_1^n \frac{1}{x} dx = \ln n$$

- $H_{n-1} = \ln n + \gamma - \epsilon$ ($\gamma \approx 0.577215$ is Euler constant)
- Since $H_n - H_{n-1} = \frac{1}{n}$ assuming ϵ to be $\frac{1}{n}$ $H_n \approx \ln n + \gamma$
- Therefore,

$$\begin{aligned} T_{average}(n) &= t_1 + t_2 + t_3(H_n - 1) \\ &\approx t_1 + t_2 + t_3(\ln n + \gamma - 1) \\ &\approx t_1 + t_3(\gamma - 1) + nt_2 + t_3 \ln n \end{aligned}$$

Best/worst case running time

- Best case running time occurs when the set is sorted in descending order - line 4 will not be executed at all, i.e.,

$$p_i = P\left[a_i > \left(\max_{0 \le j < i} a_j\right)\right] = 0$$

- Worst case running time occurs when set is sorted in ascending order - line 4 will be executed every time, i.e.,

$$p_i = P\left[a_i > \left(\max_{0 \le j < i} a_j\right)\right] = 1$$

- Therefore,

$$\begin{aligned} T_{best}(n) &= t_1 + nt_2 \\ T_{worst}(n) &= t_1 + nt_2 + \sum_{i=1,\ a_i > (\max_{0 \le j \le i} a_j)}^{n-1} t_3 \\ &= (t_1 - t_3) + (t_2 + t_3)n \end{aligned}$$

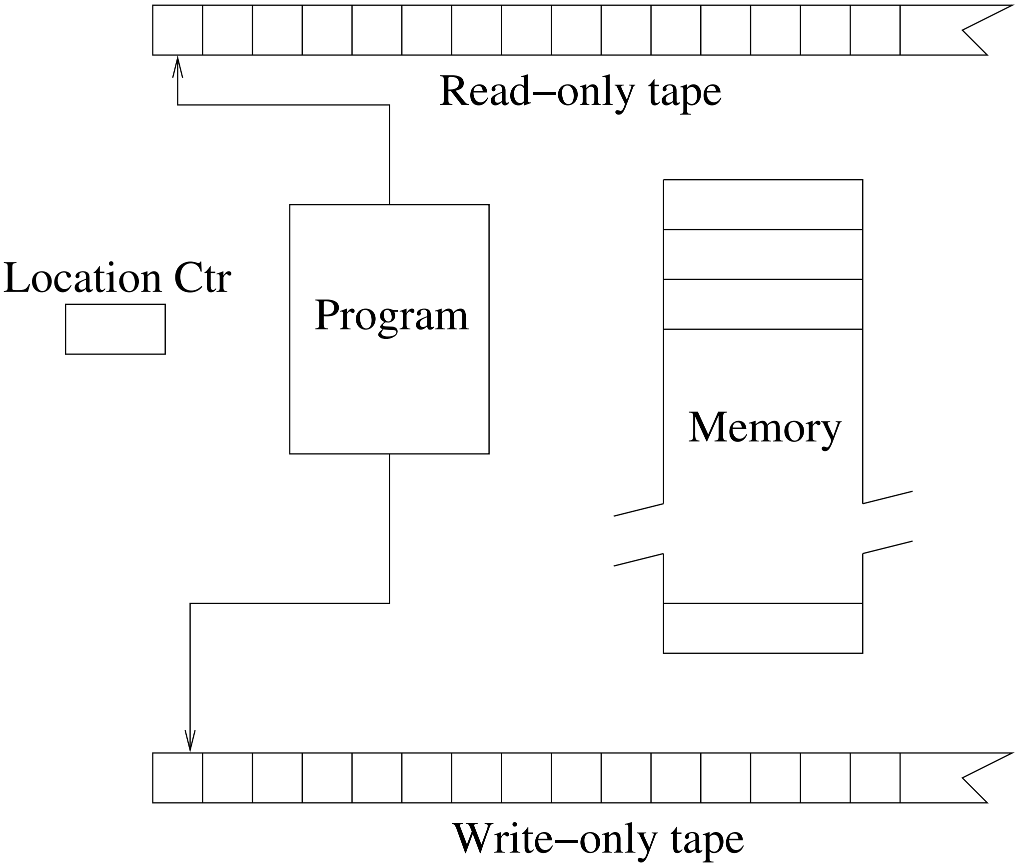
Simplifying analysis

- Each new operation has to be taken into account.
- The timing of each statement has to be carefully obtained.
- For simplifying analysis substitution have to be done.
- Specially when analyzing recursions.
- Using detail time units for analysis do not aid to capture more information about running time than we could do by assuming all operations requiring constant time unit as one unit of time.

RAM model

- A RAM or Random Access Machine consists of 5 parts
  - Store for program
  - A read-only tape of infinite length
  - A write-only tape of infinite length
  - An unbounded memory
  - A program location counter
- The memory is divided into equal-sized blocks called registers
- The register can be denoted as $r_0, r_1, r_3, \ldots$.
- The read and write tapes are divided into equal-sized square blocks each capable of holding an integer that can fit into a single computer word.
- There is a read head and also write head which advance one position to right when a symbol is read or written repsectively.
- The read/write head is controlled by the program block.

RAM model



RAM model

- The instructions are:

LOAD	operand
STORE	operand
ADD	operand
SUB	operand
MULT	operand
DIV	operand
READ	operand
WRITE	operand
JUMP	label
JGTZ	label
JZERO	label

What is an operand

- An operand can be one of the following three types
  - $=i$: *immediate*, refers directly to value of integer i
  - i: *register*, refers to value of stored in register r_i
  - $*i$: *indirect*, refers to value of stored in register r_j, where j is the value stored in register r_i.
- For example, instruction `LOAD =i` means that integer `i` is to be loaded into accumulator.
- But, instruction `STORE =i` is not a valid operation as immediate operand for `STORE` is meaningless. As `STORE` stores contents of accumulator into a memory register.
- All the operations of the above kind make them undefined and are treated as equivalent to `HALT`

Meaning of a program

- The meaning of a program is defined by its state.
- The state of a program is specified by the contents of the memory and the location counter.
- Therefore, the meaning of a program is defined by two things
  - A mapping $c : Z_0 \longrightarrow I$, the function c maps a non-negative integer i to an integer $c(i)$. Essentially $c(i)$ means contents of register r_i.
  - The contents of location counter.

The value fuction

- We associate a value function v with each operand type to make the meaning of mapping c defining the meaning of the program more clear.
- $v(=i) = i$, $v(i) = c(i)$, $v(*i) = c(c(i))$
- Thus, an instruction of the type `LOAD i` imply $c(0)$ becomes $v(i) = c(i)$, that is contents of register r_i.
- `DIV a` means $c(0) = \lfloor c(0)/v(a) \rfloor$.
- `ADD a` means $c(0) = c(0) + v(a)$.
- `READ i` means $c(i) =$ the *current input* symbol.
- `READ *i` means $c(c(i)) =$ the *current input* symbol.
- `WRITE a` means $v(a)$ is placed as on the *output* tape.

An example

- Let us consider the program for computation of the function

$$\begin{cases} n^n, \text{ if } n > 0 \\ 0, \text{ otherwise} \end{cases}$$

The algorithm for computing the above function is as follows:

```
read n to r1;
if (r1 == 0) write 0;
   else {
      r2 = r1;
      r3 = r1 - 1;
      while (r3 > 0 ) {
         r2 = r2 * r1
         r3 = r3 - 1
      }
      write r2;
   }
```

RAM program

Label	Instruction	Equivalent high level instruction
	`READ 1`	read n to r_1
	`LOAD 1`	if $(r_1 == 0)$
	`JGTZ pos`	write 0
	`WRITE =0`	
	`JUMP endif`	
`pos:`	`LOAD 1`	
	`STORE 2`	else $\{r_2 = r_1$;
	`LOAD 1`	$r_3 = r_1 - 1$;
	`SUB =1`	
	`STORE 3`	

RAM program contd.

Label	Instruction	Equivalent high-level instruction
`while:`	`LOAD 3`	`while` $(r_3 > 0)$
	`JGTZ continue`	$\{$
	`JUMP endwhile`	$r_2 = r_2 * r_1;$
`continue:`	`LOAD 2`	$r_3 = r_3 - 1;$
	`MULT 1`	$\}//$ `endwhile`
	`STORE 2`	$\}//$ `endelse`
	`LOAD 3`	
	`SUB =1`	
	`STORE 3`	
	`JUMP while`	
`endwhile:`	`WRITE 2`	`write` r_2;
`endif:`	`HALT`	

RAM program: another example

- A string s from alphabet set $\{1, 2\}$. Accept s if s has same number of 1's and 2's. The string end is marked by endmarker 0.
- High-level program is as follows:

```
d = 0;
read x;
while (x≠ 0){
   if (x≠ 1) d=d-1
      else d=d+1
   read x;
}
if (d==0) write 1;
```

RAM program: another example

Label	Instruction	Statement
	LOAD =0	d=0
	STORE 2	
	READ 1	read x
while:	LOAD 1	while (x$\neq$0)
	JZERO endwhile	
	LOAD 1	
	SUB =1	if (x $\neq$ 1)
	JZERO one	
	LOAD 2	
	SUB =1	d = d-1
	STORE 2	
	JUMP endif	

Example contd.

Label	Instruction	Statement

one:	`LOAD 2`
	`ADD =1`
	`STORE 2`
		`else d=d+1`

`endif:`	`READ 1` `JUMP while`	`read x`

`endwhile:`	`LOAD 2`
	`JZERO output`
	`HALT`
		`if (d == 0)`
		`write 1;`

`output:`	`WRITE =1` `HALT`	

Running time

- Small problem can be solved faster
- How to predict or estimate time required to solve bigger and bigger instances of a problem.
- Estimate of time is a very important aspect of the study of efficiency of an algorithm.
- It helps in comparing alternative algorithms apart from predicting growth in running time with that in the size of the problem.
- The question is: what may be common frame of reference for the comparison of running time?
- The size of the problem influences the *running time*.

Running time

- Our effort will be express running time as a monotonically increasing function of the input size.
- Then as the size of the problem grows the function will also grow in value.
- By comparing the growth rates of the function values, alternative algorithms for a problem one can determine the best suited algorithm for the expected range of input sizes of the problem.
- Estimating exact running time is preferable.
- But for large problem size the precision of the estimate is neither crucial nor worth the effort it requires.
- Lower-order terms of an expression for exact estimate of running time are dominated by the size of the input.

Running time

- Study on relative efficiency algorithms on large problem size is termed as *asymtotic* efficiency of algorithms.
- Growth rate of running time of algorithm mainly focuses on the aspect *scaling*.
- Many other important issues which indirectly become apparent during a study of this kind:
  - **Problem insight.**
  - **Better algorithms design.**
  - **Ability for program synthesis**
  - **Hardness of problem**