1 Introduction and Asympotic Analysis

Readings:

- CLRS 1.1-1.2, 2.2, 2.3.2, 3.1
- Theoretical Computer Science Cheat Sheet (TCSCS)

1.1 Motivation: Three Algorithms for Sorting

Selection sort:

```
for i:=1 to n
| min:=i;
| for j:=i+1 to n
| | if a[min]>a[j] min:=j;
| tmp:=a[i]; a[i]:=a[min]; a[min]:=tmp;
```

Merge sort:

```
function sort(from,to)
| if (from>to)
| | mid:=floor(from+to/2);
| | sort(from,mid); sort(mid+1,to);
| | merge(from,mid,to);

function merge(from,mid,to)
| copy a[from..mid] to a new array b
| copy a[mid+1..to] to a new array c
| add infinity to both b and c as the last element
|
| k:=1; m:=1;
| for j:=from to to
| | if b[k]<c[m] then
| | | a[j]:=b[k]; k++;
| | else
| | | a[j]:=c[m]; m++;
```

Counting sort:

Assumption: all numbers in the array are between 1 and 1000

```
clear array count[1..1000];
for i:=1 to n
| count[a[i]]++;
k:=0;
for i:=1 to 1000
| for j:=1 to count[i];
| | a[k]:=i; k++;
```

How do we find out which of these algorithms is the best?

1.2 Analyzing Running Time of Algorithms

- The running time $T_A(x)$ of an algorithm A for a particular input x is the $\boxed{\text{time}}$ that the algorithm requires to solve the input x.
- The **worst-case running time** $T_A(n)$ of an algorithm A is a function of the size n of the input, where $T_A(n)$ is the *largest time* required to solve an input of size n:

$$T_A(n) = \max\{T_A(x) \mid |x| = n\}.$$

- The **average-case running time** $T_A^{(\text{avg})}(n)$ of an algorithm A is a function of the size n of the input, where $T_A(n)$ is a *average of times* required to solve all inputs of size n:

$$T_A^{(\text{avg})}(n) = \text{avg}\{T_A(x) \mid |x| = n\}.$$

$\boxed{\textbf{How to measure time?}}$ The actual execution time depends on many factors, specific to a particular implementation of the algorithm, as well as properties of a particular computer on which we run the implementation (exact speed of the processor, disk, memory, amount of cache memory, ...). To simplify theoretical analysis, we need to abstract away from these issues. Thus, instead of measuring real time, we will *count elementary operations.*

Elementary operation: This is an operation whose time can be bounded by a constant that depends only on the implementation of the operation (either in hardware or software) and not on the inputs to the operation.

- **Elementary operations:** simple arithmetic operations, comparisons, program flow control operations, etc.
- **Not elementary operations:** maximum in an array of numbers, does string contain a given substring?, concatenation of two strings, factorial, etc. (beware: many programming languages offer constructs that are not elementary operations)

1.3 Random Access Machine (RAM) Model

Sometimes, we need to formalize a theoretical model of a computer, to determine what is an elementary operation, and what is not. Commonly used model in computer science is *random access machines.*

Here is the overview of the model (you will see more precise formulation in CS341):

- The computer's memory consists of unbounded number of memory cells.
- Each memory cell holds one of the following: an integer, a real number, or a character of a string.
- The following operations take one unit of time: addition, subtraction, multiplication, division, remainder, floor, ceiling, memory cell access (load, store, copy), program flow control operations (branching, calling subroutines)
- Other operations should be described in terms of operations above, and thus may not be constant time.
- Instructions are executed in sequence (no parallelism)

1.4 Asymptotic Notation for Comparing Algorithms

Since we are now measuring time by counting elementary operations rather than measuring actual running time on a computer, and each of these operations can take different time in reality, it does not make sense to compare absolute counts. Instead, we need a way to compare the algorithms where constants do not matter.

Definition 1 *Function $f(n)$ is in $O(g(n))$ iff there exist $c > 0$ and $n_0 > 0$ such that:*

$$(\forall n > n_0)(0 \leq f(n) \leq cg(n))$$

Notation: $f(n) \in O(g(n))$ or $f(n) = O(g(n))$.

Now we can analyze the three algorithms for sorting:

- **Selection sort:**

```
for i:=1 to n
| min:=i;
| for j:=i+1 to n
| | if a[min]>a[j] min:=j;
| tmp:=a[i]; a[i]:=a[min]; a[min]:=tmp;
```

Each line of this pseudocode is constant time. Therefore, we can count each line as an elementary operation. We can replace the loops with sums as follows:

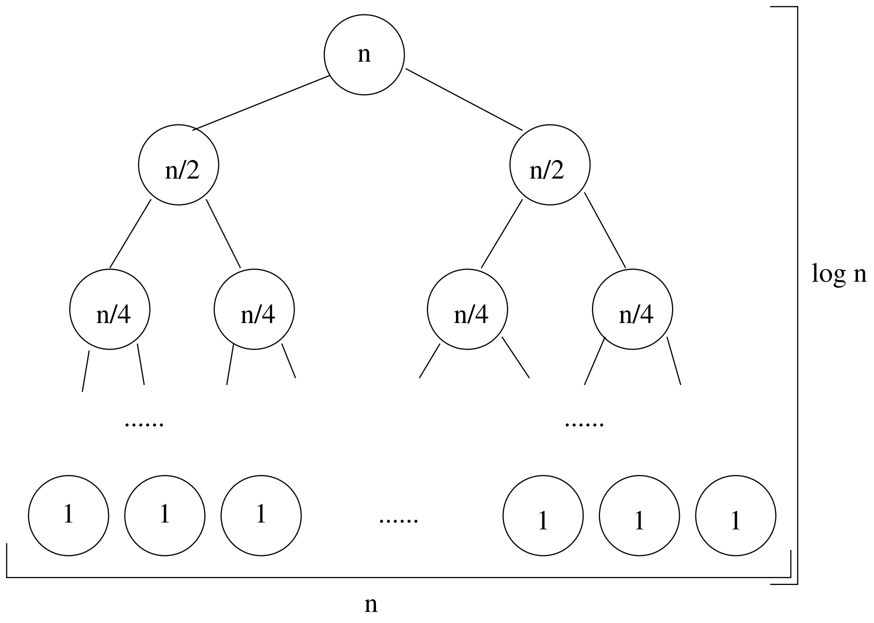
$$\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}1\right) = n + \sum_{i=1}^{n}(n-i) \leq n^2 + n$$

Therefore, the running time is $\boxed{O(n^2).}$

- **Merge sort:**

```
function sort(from,to)
| if (from>to)
| | mid:=floor(from+to/2);
| | sort(from,mid); sort(mid+1,to);
| | merge(from,mid,to);
```

First of all, function merge has clearly running time of $O(\text{to} - \text{from})$ (the analysis is similar to the analysis of selection sort above). We can visualize the running time of the recursive function sort as a tree. Each node of the tree will represent one call of the function, and the number inside each node represents the running time of that function, excluding running time needed for the recursive call of the function (which are accounted for by children nodes in the tree):



Observe that the running times associated with nodes at each level sum to n. Since there are $\log n$ levels, the overall running time is $\boxed{O(n \log n).}$

- **Counting sort:**

```
1. clear array count[1..1000];
2. for i:=1 to n
3. | count[a[i]]++;

4. k:=0;
5. for i:=1 to 1000
6. | for j:=1 to count[i];
7. | | a[k]:=i; k++;
```

Line 1 takes $O(1)$ time (1000 is a constant). A cycle at lines 2-3 takes clearly $O(n)$ time. Line 7 is executed $\sum_{i=1}^{1000} \text{count}[i] = n$ times. Therefore, the counting sort takes $O(n)$ time.

This analysis seems to suggest that "counting sort" is better than "merge sort" which is better than "selection sort". Does this analysis have any relationship to the actual running times on a real computer?

		Counting sort $O(n)$	**Merge sort** $O(n \log n)$	**Selection sort** $O(n^2)$	**Exponential algorithm** $O(2^n)$
Time to	10	ε	ε	ε	ε
solve a	50	ε	ε	ε	2 weeks
problem	100	ε	ε	ε	2800 univ.
of size	1000	0.01s	ε	ε	—
...	10000	0.01s	0.03s	0.27s	—
	100000	0.06s	0.15s	26.5s	—
	1 mil.	0.74s	1.6s	44.2m	—
	10 mil.	7.4s	16.5s	3.1d	—
Max size	1s	1.4 mil.	720000	19400	33
problem	1m	81 mil.	34 mil.	150000	39
solved in	1d	117 bil.	35 bil.	6 mil.	49
Increase in	+1	—	—	—	×2
time if n	×2	×2	×2+	×4	—
increases					

(Some values were measured on Pentium 4 2Ghz computer, some of the values were estimated from other measurements. The exponential algorithm is a fictitional algorithm (just for comparison to others) ε means under 0.01s. "univ" means time elapsed from the start of universe, as estimated by physicists.)

- Even with today's fast processors, better algorithms matter.
- Asymptotic analysis allows us to easily analyze and compare algorithms without considering details specific to a particular computer.
- For a single problem there can be several solutions with different complexities.

1.5 Some Properties of O Notation

The following claims can be proven directly from the definition. Attempting this on your own is a good exercise.

- if $f(n) \in O(g(n))$ and $c > 0$ is a constant
  then $cf(n) \in O(g(n))$

  (for example: $100n \in O(n)$)

- **Maximum rule.** If $t(n) \in O(f(n) + g(n))$
  then $t(n) \in O(\max(f(n), g(n)))$

  (for example: if a loop that takes $O(n)$ time is followed by a section of codes takes $O(n^2)$ time, the whole code takes $O(n^2)$ time)

- **Transitivity.** If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$
  then $f(n) \in O(h(n))$

- $n^k \in O(a^n)$ for all constants $k > 0$ and $a > 1$

  (any polynomial algorithm is better than exponential, no matter what is the degree/base)

- Similarly: $\log^k n \in O(n^a)$ for all constants $k > 0$ and $a > 0$

  (logarithmic is better than polynomial)

- $n^k \in O(n^\ell)$ for all constants $k > 0$ and $\ell \geq k$

  (lower order polynomial algorithms are better than higher order polynomials)

- if $f(n) \in O(f'(n))$ and $g(n) \in O(g'(n))$ then $f(n)g(n) \in O(f'(n)g'(n))$

Examples:

- $3.8n^2 + 2.6n^3 + 10n \log n \in O(n^3)$ (we use as simple a form as possible)
- $10^{100}n \in O(n)$
- $(n + 1)! \in O(n!)$ true or <u>false</u>?
- $2^{2n} \in O(2^n)$ true or <u>false</u>?
- $n \in O(n^{10})$ <u>true</u> or false?

1.6 More Asymptotic Notations

Notation	**Definition**	**Analogy to arithmetic comparisons**
$f(n) \in O(g(n))$	There exists $c > 0$ and $n_0 > 0$ s.t. $(\forall n > n_0)(0 \leq f(n) \leq cg(n))$	$\leq$
$f(n) \in \Omega(g(n))$	There exists $c > 0$ and $n_0 > 0$ s.t. $(\forall n > n_0)(f(n) \geq cg(n) \geq 0)$	$\geq$
$f(n) \in \Theta(g(n))$	$f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$	$=$
$f(n) \in o(g(n))$	For any $c > 0$ there exists $n_0 > 0$ s.t. $(\forall n > n_0)(0 \leq f(n) < cg(n))$	$<$
$f(n) \in \omega(g(n))$	For any $c > 0$ there exists $n_0 > 0$ s.t. $(\forall n > n_0)(f(n) > cg(n) \geq 0)$	$>$

Exercise: Can rules similar to the ones we mentioned in case of O notation be applied in case of the other notations?

- if $f(n) \in \omega(g(n))$ then $f(n) \notin O(g(n))$
- if $f(n) \in O(g(n))$ then $f(n) \notin \omega(g(n))$
- but there are functions where $f(n) \notin O(g(n))$ and $f(n) \notin \Omega(g(n))$

1.7 How to Prove Negative Results? $f(n) \notin O(g(n))$

Definition:

$$f(n) \in O(g(n)) \Leftrightarrow (\exists c > 0)(\exists n_0 > 0)(\forall n > n_0)(0 \leq f(n) \leq cg(n))$$

Negation:

$$f(n) \notin O(g(n)) \Leftrightarrow (\forall c > 0)(\forall n_0 > 0)(\exists n > n_0)(f(n) < 0 \text{ or } f(n) > cg(n))$$

Example: $(n+1)! \notin O(n!)$

We know $(n+1)! = (n+1) \cdot n!$. For any $c > 0$ and $n_0 > 0$, take $n = \lceil c \rceil \lceil n_0 \rceil$. Then:

$$(\lceil c \rceil \lceil n_0 \rceil + 1)(\lceil c \rceil \lceil n_0 \rceil)! > c(\lceil c \rceil \lceil n_0 \rceil)!$$

Note: The negation is not identical to the definition of $\omega(g(n))$.