

Unit 1: Introduction



1.1 Introduction to Algorithm Analysis

Abstract Runtime (ART)

- What should we analyze in an algorithm?
 - Correctness.
 - Memory use.
 - **Performance (in terms of *running time*).**
- How to measure performance?
 - With a clock (plain running time)
 - ***Intuitive but problematic ...***
 - **By analyzing the associated pseudo-code:
abstract runtime (ART)**

How to measure ART?

- Option1: Count the number of statements (lines ending with ; which are not function calls) that the algorithm runs given an input I.
 - This option is complex
 - It depends on how the pseudocode is written
 - It provides unnecessary information
- In this option we assign 1 ART unit to each statement.
- Let us consider an example...

Examples of measuring ART:

□ Example: matrix multiplication (pseudocode)

```
matrix MM(matrix A, matrix B, dim N)
  for i=1 to N:
    for j=1 to N:
      c[i, j] = 0.;
      for k=1 to N:
        c[i, j] += a[i, k] * b[k, j];
  return c
```

Calculating MM ART

$$ART_{MM}(A, B, N) = \sum_{i=1}^N ART(iter(i)) =$$

$$\sum_{i=1}^N \sum_{j=1}^N ART(iter(j)) =$$

$$\sum_{i=1}^N \sum_{j=1}^N \left(1 + \sum_{k=1}^N ART(iter(k)) \right)$$

$$\leq \sum_{i=1}^N \sum_{j=1}^N (1 + N)$$

$$= N^2(1 + N) = N^3 + N^2$$

Thus, we get an ART in the form $f(N) = f(\text{input size})$

Calculating ART: SelectSort

```
void SelectSort(array T, ind F, ind L)
    i=F;
    while i<L:
        min=i ;
        for j = i+1 to L :
            if T[j]<T[min] :
                min=j ;
        swap(T[i],T[min]) ;    i++;
```

□ Shorter version:

```
void SelectSort(array T, ind F, ind L)
    for i = F to L-1:
        min= min(T, i, L)
        swap(T[i],T[min]);
```

SelectSort sorting strategy

- Example. SelectSort
- Input: $T=[4,3,2,1]$, $F=1$, $L=4$.

Iteration	Operations	Array/Table
$i=1$	$\text{min}=4$ $\text{swap}(T[1],T[4])$	$[1,3,2,4]$
$i=2$	$\text{min}=3$ $\text{swap}(T[2],T[3])$	$[1,2,3,4]$
$i=3$	$\text{min}=3$ $\text{swap}(T[3],T[3])$	$[1,2,3,4]$

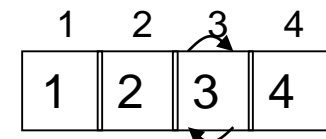
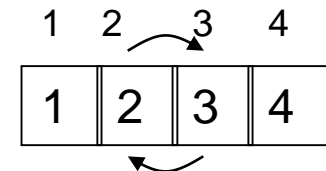
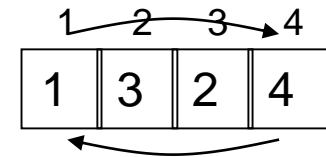
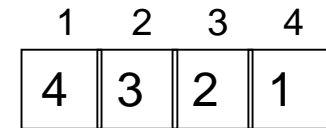
SelectSort sorting strategy II

□ Now in more detail:

Input: $T=[4,3,2,1]$, $F=1$, $L=4$.

Loop 1

- $i=1$
 - Loop 2
 - $\min = 1$
 - $j=2$ $T[2]<T[1]$? Yes $\rightarrow \min=2$
 - $j=3$ $T[3]<T[2]$? Yes $\rightarrow \min=3$
 - $j=4$ $T[4]<T[3]$? Yes $\rightarrow \min=4$
 - $\min=4$, $\text{swap}(T[1],T[4])$
- $i=2$
 - Loop 2
 - $\min = 2$
 - $j=3$ $T[3]<T[2]$? Yes $\rightarrow \min=3$
 - $j=4$ $T[4]<T[3]$? No $\rightarrow \min=3$
 - $\min=3$, $\text{swap}(T[2],T[3])$
- $i=3$
 - Loop 2
 - $\min = 3$
 - $T[4]<T[3]$? No $\rightarrow \min=3$
 - $\min=3$, $\text{swap}(T[3],T[3])$
- $i=4$ End ($i=U$)



SelectSort ART

- SelectSort works with two nested loops

```
void SelectSort(Array T, ind F, ind L)
```

```
    i=F;
```

```
    while i<L:
```

Loop 1

Loop 2

```
        min=i;
```

```
        for j = i+1 to L:
```

```
            if T[j]<T[min] :
```

```
                min=j;
```

```
        swap(T[i],T[min]);
```

```
        i++;
```

Let's try to calculate the associated ART

SelectSort ART II

■ Calculating $ART_{\text{Selectsort}}(T;F,L)$:

```
void SelectSort(Tabla T, ind F, ind L)
```

```
    i=F;
```

```
    while i<L:
```

← Loop from F to L-1

```
        min=i;
```

```
        for j=i+1 to L:
```

← Loop from i+1 to L

```
            if T[j]<T[min]:
```

```
                min=j;
```

```
        swap(T[i],T[min]);
```

```
        i++;
```

Loop 1
L-F
iterations

Loop 2
L-i
iterations

It seems that the number of loop cycles and their size determines ART

SelectSort ART III

Noting that $N=U-P+1$ is the size of array T

$$\begin{aligned}
 ART_{ss}(T, F, L) &= 1 + \sum_{i=F}^{L-1} ART(iter(i)) = 1 + \sum_{i=F}^{L-1} (4 + tae_{loop\ 2}(T, i, L)) \\
 &\leq 1 + \sum_{i=F}^{L-1} (4 + 3(L-i)) = 1 + 4(L-F) + 3 \sum_{j=1}^{L-F} j \\
 &= 1 + 4(L-F) + 3(L-F+1)(L-F)/2 \\
 &= 1 + 4(N-1) + 3(N-1)N/2 \\
 &= 3N^2/2 + 5N/2 - 3
 \end{aligned}$$

$$N = L - F + 1$$

$$\sum_{i=1}^N i = (N+1)N/2$$

This leads to an ART in the form $f(N)$

There are issues open for discussion, mainly the coefficients

For example: is the swap ART 1 or 3?

How to simplify and normalize ART?

- **Observation: MM and SS ARTs are dominated by the innermost loop**

- **Option 2.**
 - Define a **basic operation** and count the number of times that algorithm **A** runs it on an input **I** ($n_A(I)$)
 - The abstract running time of **A** on input **I** is the **number of basic operations** that **A** runs on **I**: $ART_A(I) = n_A(I)$.

- **Basic operation:**
 - We can find it in the innermost loop of the algorithm (it is the operation that is executed most times)
 - It must be “representative” of the algorithm (also of other algorithms that solve the same problem).

Revisiting MM pseudocode

□ Example: matrix multiplication (pseudocode)

```
matrix MM(matrix A, matrix B, dim N)
```

```
  for i=1 to N:
```

```
    for j=1 to N:
```

```
      c[i, j] = 0.;
```

```
      for k=1 to N:
```

```
        c[i, j] += a[i, k] * b[k, j];
```

```
  return c
```

□ Basic operation (OB): *

□ $n_{MM}(A, B, N) = N^3$

Revisiting SelectSort pseudocode

```
void SelectSort(Array T, ind F, ind L)
    i=F;
    while i<L:
        min=i;
        for j = i+1 to L:
            if T[j]<T[min]:
                min=j ;
        swap(T[i],T[min]) ;
        i++;
```

SelectSort Basic Operation

■ Basic Operation

- It must be in the innermost loop.
- It must be “representative” of the algorithm.

■ A good candidate for BO of SelectSort is:

if $T[j] < T[\text{min}]$:

- This operation is called “key comparison” (KC).
- Indeed, it is located in the innermost loop.
- It is representative of SelectSort and of many other sorting algorithms.

SelectSort ART analysis

■ Calculating $n_{\text{SelectSort}}(T, F, L)$

```
void SelectSort(array T, ind F, ind L)
```

```
    i=F;
```

```
    while i<L:
```

← loop from F to L-1

```
        min=i;
```

```
        for j=i+1 a L:
```

← loop from i+1 to L

```
            if T[j]<T[min]:
```

```
                min=j;
```

```
        swap(T[i],T[min]);
```

```
        i++;
```

loop 1
L-F
iterations

loop 2
L-i
iterations

$$\begin{aligned}
 n_{\text{SelectSort}}(T, 1, N) &= \sum_{i=1}^{N-1} n_{\text{loop } 2}(T, i, N) = \sum_{i=1}^{N-1} \sum_{j=i+1}^N 1 = \sum_{i=1}^{N-1} (N-i) = \sum_{j=1}^{N-1} j \\
 &= \frac{N(N-1)}{2} = \frac{N^2}{2} - \frac{N}{2}
 \end{aligned}$$

In summary ...

- We have been able to evaluate a first version of Selectsort ART.
- The result, $ART=N^2$ is reasonable, since it takes into account the size of the array and the number of loops.
- The BO eliminates the ambiguities associated with statement counting.
- ART depends on the number of loops and on their size.
- The analysis is easy when the loop's ART does not change in each iteration.

Example: Binary Search

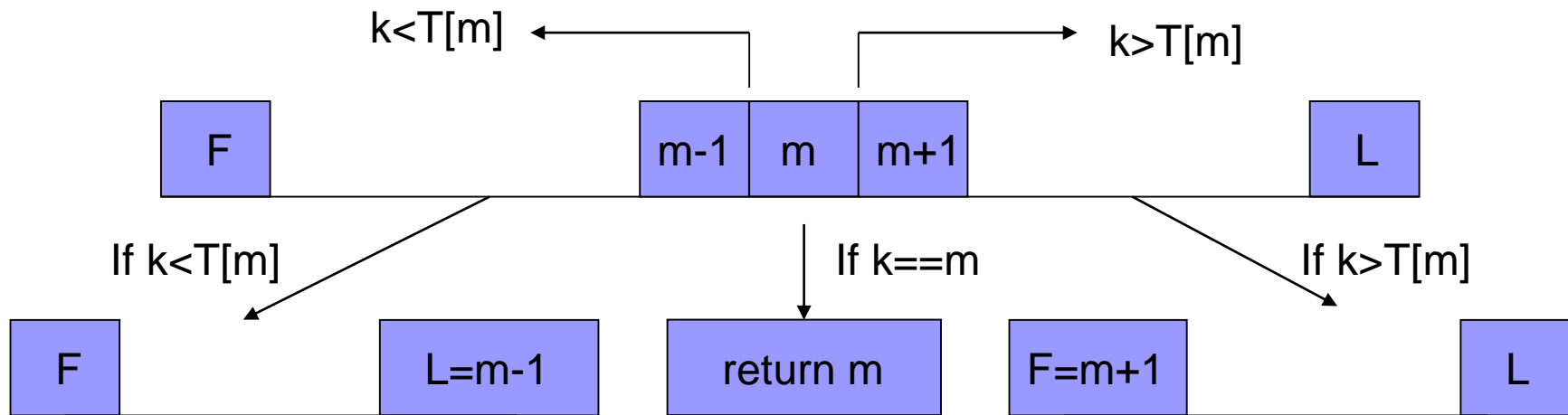
□ Binary Search pseudocode

```
ind BSearch(Tabla T, ind F, ind L, key k)
  while  $F \leq L$  :
     $m = (F + L) / 2$ ;
    if  $T[m] == k$  :
      return m;
    else if  $k < T[m]$  :
       $L = m - 1$ ;
    else :
       $F = m + 1$ ;
  return error;
```

Important: BSearch only works if array T is already sorted.

How does BSearch work?

- In each iteration, the algorithm either returns or the search in the array is reduced:



- The size of the searchable array is reduced approximately in half after each iteration.
- If the algorithm exits from the cycle $F \leq L$ because $F > L$, it means that the searched key k is not in T .

OB para BSearch

- Since here we only have one loop, a good candidate for BO is the following comparison operation:

```
if T[m]==k :
```

```
.....
```

```
else if k < T[m] :
```

```
.....
```

```
else :
```

```
.....
```

- Although there are two comparisons, we only count 1 ART unit
- Again, this is a key comparison operation.

BSearch ART analysis

- $n_{\text{BSearch}}(T, F, L, k) = \text{number of iterations}(F < L) \leq \text{number of times that a number } N \text{ can be divided by } 2.$
- After each iteration, the size of the resulting array is less than half the previous one, thus

$$N \rightarrow \frac{N}{2} \rightarrow \frac{N}{2^2} \rightarrow \dots \rightarrow \boxed{\frac{N}{2^k} \leq 1}$$

K = maximum number of divisions by 2 = maximum number of iterations

$$2^{k-1} < N \leq 2^k \Rightarrow k = \lceil \log_2 N \rceil \geq n_{\text{BSearch}}(T, 1, N, k)$$

- **Exercise:** Calculate BSearch ART counting all sentences instead of only counting the BO.
- **Solution** (arguable): $n_{\text{BSearch}} \leq 5 \lceil \log_2 N \rceil + 2$

Observations on ART

- Observation 1: It seems possible that for any algorithm **A**, we can assign its inputs **I** an **input size** ($\tau(I)$) and find a given function of **N**, $f_A(N)$ such that

$$ART_A(I) = n_A(I) \leq f_A(\tau(I))$$

- In the discussed examples

A	I	τ	f_A
SelectSort	(T,F,L)	$L-F+1$	$N^2/2 - N/2$
BBin	(T,F,L,k)	$L-F+1$	$\lceil \lg N \rceil$

- Observation 2: when calculating ART, there is always a **dominant term**; the rest contribute less.

Observations on ART

- Observation 3: ART allows
 - generalizing running times as a function of the input size.
 - estimating the real running time.
- Example: SelectSort with I such that $\tau(I)=N$ and I' such that $\tau(I')=2N$
 - $ART_{SSort}(I) = N^2/2 + \dots$
 - $ART_{SSort}(I') = (2N)^2/2 + \dots = 4N^2/2 + \dots \sim 4ART_{SSort}(I)$
- In general, if $\tau(I')=kN$, we get $ART_{SSort}(I') \sim k^2 ART_{SSort}(I)$
- If it takes 1 second to sort an array with $N=1000$, then

Size	1000	2000	10000	100000
Time taken	1s	4s	100s	10000s

Example: the “bubble” method

- Sorting with a bubble (BubbleSort_v1).

```
BubbleSort_v1(Array T, ind F, ind L)
  for i = L to F+1:
    for j = F to i-1:
      if T[j]>T[j+1]:
        swap(T[j],T[j+1]);
```







- Index j: bubble that tries to float up. If it cannot go up any more, we change the bubble.
- After each iteration in i, the largest number in the subarray is in position i.
- The array is sorted from right to left.

Bubblesort operation

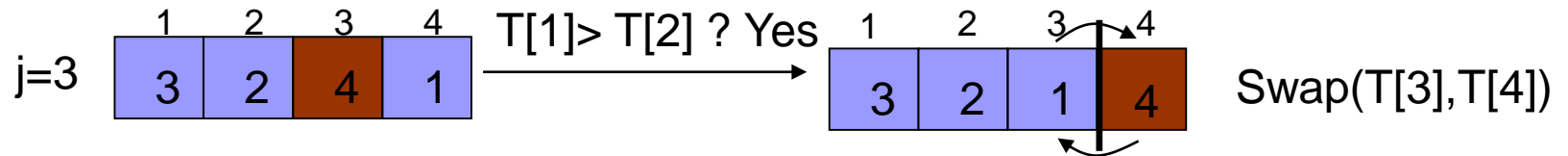
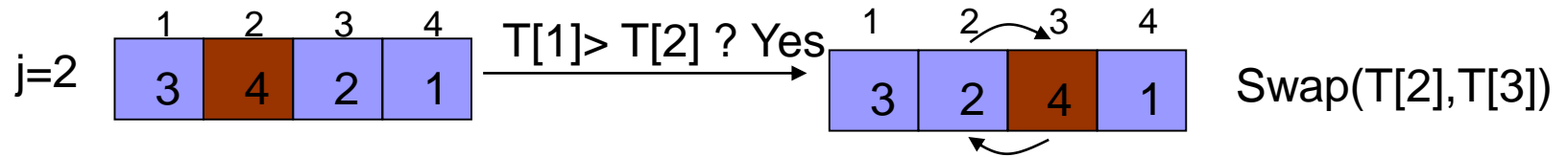
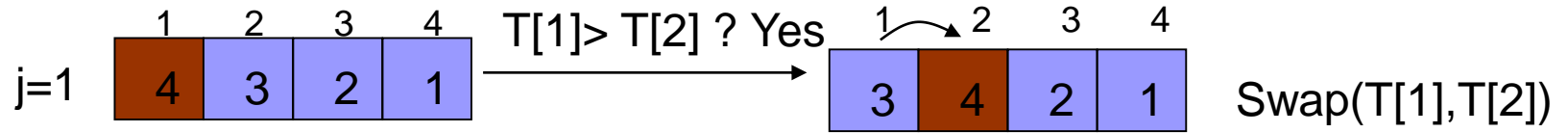
■ BubbleSort_v1

Input: $T=[4,3,2,1]$, $F=1$, $L=4$.

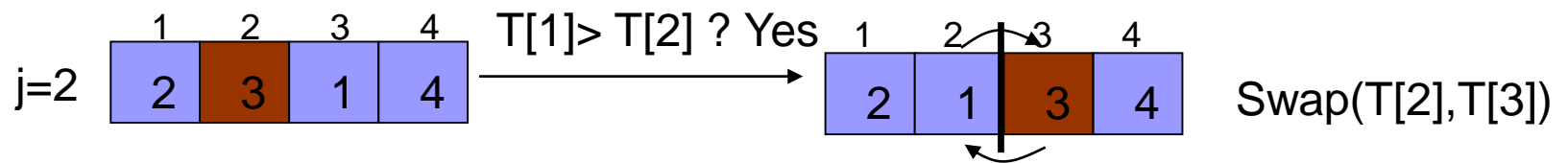
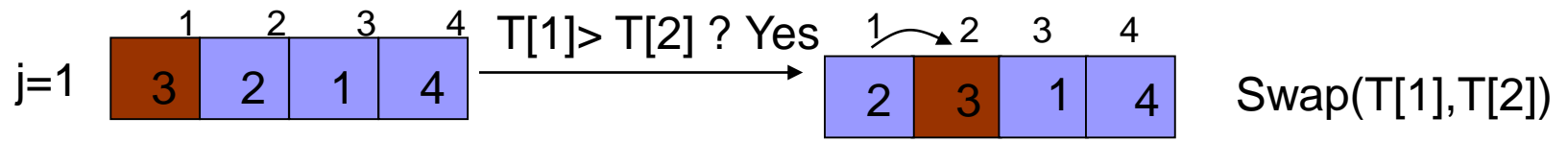
Bubble

i	j	Operations	Array
4	1	$T[1] > T[2]$? Yes swap($T[1], T[2]$)	3, 4, 2, 1 
4	2	$T[2] > T[3]$? Yes swap($T[2], T[3]$)	3, 2, 4, 1 
4	3	$T[3] > T[4]$? Yes swap($T[3], T[4]$)	3, 2, 1, 4 
3	1	$T[1] > T[2]$? Yes swap($T[1], T[2]$)	2, 3, 1, 4 
3	2	$T[2] > T[3]$? Yes swap($T[2], T[3]$)	2, 1, 3, 4 
2	1	$T[1] > T[2]$? Yes swap($T[1], T[2]$)	1, 2, 3, 4 

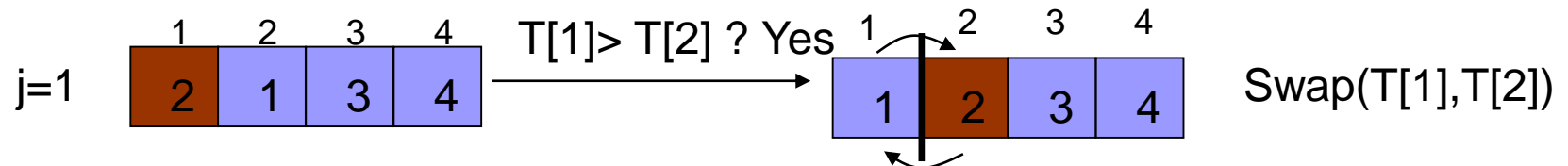
i=4



i=3



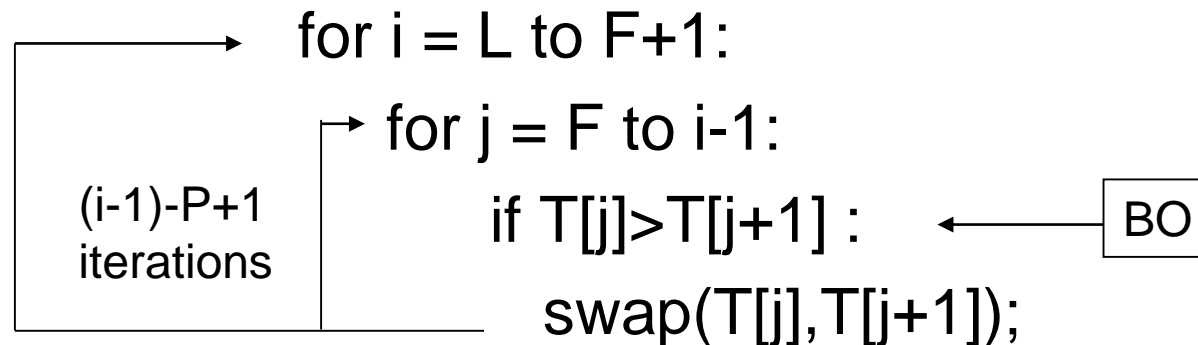
i=2



BubbleSort ART

- Basic operation (BO)? Key comparison !!!
- Sorting with a bubble (BubbleSort_v1).

BubbleSort_v1(Array T, ind F, ind L)



- Thus,

$$n_{BSort_v1}(T, 1, N) = \sum_{i=2}^N \sum_{j=1}^{i-1} 1 = \sum_{i=2}^N (i-1) = \sum_{i=1}^{N-1} i = \frac{N^2}{2} - \frac{N}{2}$$

Observations on BubbleSort ART

- BubbleSort_v1's ART (also SelectSort's) **DOES NOT** depend on the input (it always yields $N^2/2 - N/2$).
 - It takes the same time to sort an unsorted array that a sorted array!!!

- There is nothing we can do in SelectSort:
 - There is no way to know whether T is already sorted

Improving the Bubble

- In BurbujaSort_v1 it is possible to know whether T is sorted or not
- Example: $T = [1\ 2\ 3\ 4]$
 - In the first iteration no swaps are done because the subarray is sorted.
- We can add a flag to test whether swaps are done in the innermost loop.
- If swaps are not done, the subarray is already sorted and also the full array.
- We can then end the algorithm and thus improve its associated ART.

BubbleSort V2 ART

- Sorting with a bubble (BubbleSort_v2).

```
BubbleSort_v2(Array T, ind F, ind L)
```

```
Flag=1; i=L;
```

```
while (Flag==1 && i≥F+1) :
```

```
Flag=0;
```

```
for j = F to i-1 :
```

```
if T[j]>T[j+1] :
```

```
swap(T[j],T[j+1]); Flag=1;
```

```
i--;
```

- Now we have $n_{BSort_v2}([1,2,\dots,N])=N-1$ and in the worst case we keep having $n_{BSort_v2}(T,1,N) \leq N^2/2 - N/2$

But ...

- We are working with individual algorithms...
- We want to compare algorithms against each other.
- Q: How can we compare two algorithms?

ART algorithm comparison

- It only makes sense on “similar” algorithms
 1. that solve the same type of problem (sorting, search)
 2. that have the same basic operation
- We can group algorithms in families **F** that meet conditions 1 y 2.
- Example:
 $F = \{\text{Sorting algorithms that use key comparisons}\}$
 $= \{\text{InsertSort, SelectSort, BubbleSort, QuickSort, MergeSort,}\}$

ART algorithm comparison II

- We will use the following method:
 1. For a given algorithm $A \in F$, we look for $f_A(N)$ such that
$$n_A(I) \leq f_A(\tau(I))$$
where $\tau(I)$ is the input size.
 2. We say that A_1 is better (in terms of ART) than A_2 if
$$f_{A_1}(N) \text{ is "less" than } f_{A_2}(N)$$
- The comparison only makes sense for “large” inputs = **asymptotic** comparison
- We will only compare $f_{A_1}(N)$ and $f_{A_2}(N)$ when N is large (i.e., when $N \rightarrow \infty$):

In this section we have learnt...

- A basic measure of running time **algorithm efficiency** in terms of running time.
- The concept of **basic operation (BO)**
- The operation of several simple algorithms (BSearch, BubbleSort, SelectSort).
- The calculation of the ART performance of the above mentioned simple algorithms as the number of BOs.
- How to approach the task of **algorithm comparison**.

Tools and techniques to work on...

- Summation notation.
- Estimation of the number of iterations in loops.
- A good knowledge of the operation of the algorithm to be analyzed.
- Exercises to solve: those recommended in section 1 in the exercise and problem sheets (at least !!!).



1.2 Estimation of function growth

Asymptotic comparison of functions: o

- **Definition 1** ($f=o(g)$, 'little o' notation $f \ll g$): If f and g are positive functions, we say that **$f=o(g)$** if

$$\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = 0$$

- Examples:

$$f=N^k, \quad g=N^{k+\varepsilon}$$

$$f=\log(N), \quad g=N^\varepsilon$$

$$f=(\log(N))^k, \quad g=N^\varepsilon$$

(exercise; prove it using L'Hôpital's rule)

Asymptotic comparison of functions: O

- **Definition 2** ($f=O(g)$, ‘big O ’ notation $f \leq g$): **$f=O(g)$** if N_0 and $C \geq 0$ exist such that for all $N \geq N_0$,
 $f(N) \leq Cg(N)$

- Example:

$$f=N^2, \quad g=N^2+\sqrt{N}$$

And also **$g = O(f)$**

- Intuitive interpretation: g is larger or equal than f
“eventually and with some help”

Observations on $f=O(g)$

- $f=o(g) \Rightarrow f=O(g)$

- The reciprocal is false:

$$f=O(g) \not\Rightarrow f=o(g)$$

- Constants “do not matter” in O , i.e., if

$$f=O(g) \Rightarrow f=O(kg) \text{ with } k > 0.$$

- If $f=O(g)$ and $h=o(g)$ then $f=O(g+h)$, since $g+h = O(g)$

- Adding other lower terms like $f=O(N^2+N)$ does not add precision.

- It is enough expressing that $f=O(N^2)$

Asymptotic comparison of functions: Θ

- Definition 3 ($f=\Omega(g)$, $f \geq g$)

$$f=\Omega(g) \text{ if } g=O(f)$$

- Definition 4 ($f=\Theta(g)$, $f = g$)

$$f= \Theta(g) \text{ if } f=O(g) \text{ and } f=\Omega(g)$$

- Note that

$$f=\Theta(g) \Leftrightarrow g=\Theta(f)$$

- Furthermore, if $\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = L \neq 0$, then $f=\Theta(g)$

Asymptotic comparison of functions: \sim

- Definition 5 ($f \sim g$): The function f is said to be asymptotically equivalent to g if

$$\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = 1$$

- Example

$$f(N) = N^2 + \sqrt{N} + \log N, \quad g(N) = N^2$$

- Observation

$$f \sim g \Rightarrow f = \Theta(g)$$

The reciprocal is false

Asymptotic comparison of functions: $f=g+O(h)$

- Definition 6 ($f=g+O(h)$)

If $h=o(g)$, we say that **$f=g+O(h)$** if $|f-g|=O(h)$

- Example

$$f(N) = N^2 + \sqrt{N} + \log(N), \quad g(N) = N^2$$

$$\text{Then } \mathbf{f = g + O(\sqrt{N})}$$

Asymptotic comparison of functions

- We have a scale of **decreasing precision**:
 - If $f=g+O(h)$ then $f\sim g$
 - If $f\sim g$ then $f=\Theta(g)$
 - If $f=\Theta(g)$ then $f=O(g)$
- But the reciprocals are false.

Asymptotic comparison of functions II

Furthermore:

- If $f=O(g)$ and $f' = O(g')$, then

$$f + f' = O(g + g')$$

$$f f' = O(g g')$$

- Thus, if $P(N)$ is a k^{th} -grade polynomial we can write: $P(N)=a_k N^k+O(N^{k-1})$

Growth of arithmetic and geometric series

We have already seen that

$$f(N) = S_N = \sum_{i=1}^N i = \frac{N(N-1)}{2} = \frac{N^2}{2} + O(N)$$

Geometric series (**very important in AA!!**)

$$S_N = \sum_{i=1}^N x^i = \frac{x^{N+1} - x}{x - 1} = \frac{LR - F}{R - 1}$$

Observations:

- If $x = 1$ $S_N = N$
- If $x > 1$ $S_N = \Theta(x^N)$
- If $x < 1$ $S_N = \frac{x - x^{N+1}}{1 - x} \xrightarrow{N \rightarrow \infty} \frac{x}{1 - x}$

Derivative function growth

- Calculation using derivatives

$$S_N = \sum_{i=1}^N ix^i = x \frac{d}{dx} \sum_{i=1}^N x^i = \Theta(Nx^N)$$

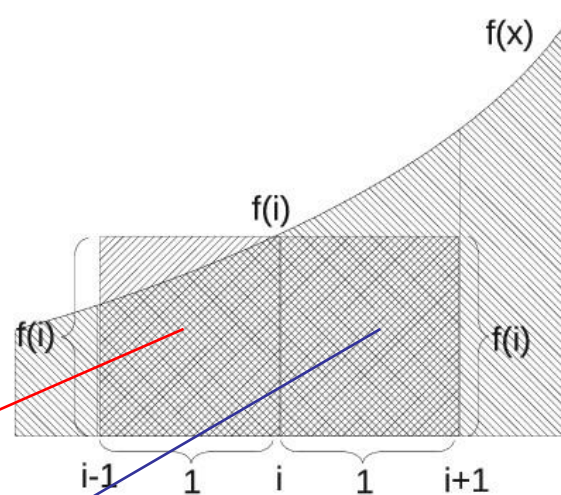
- Cubic series

$$S_N = \sum_{i=1}^N i^2 = \frac{N(N+1)(2N+1)}{6} = \frac{N^3}{3} + O(N^2)$$

- We are more interested in the growth rather than a closed formula.
- Can we simplify?

Estimation of summation growth

- What can we do when we cannot have a closed-form for a summation $S_N = \sum_{i=1}^N f(i)$?
- We can **approximate the summation by an integral**.



$f(x)$ increasing function

$$\int_{i-1}^i f(x)dx \leq f(i) \leq \int_i^{i+1} f(x)dx \Rightarrow \sum_{i=1}^N \int_{i-1}^i f(x)dx \leq \sum_{i=1}^N f(i) \leq \sum_{i=1}^N \int_i^{i+1} f(x)dx \Rightarrow$$

$$\int_0^N f(x)dx \leq S_N \leq \int_1^{N+1} f(x)dx$$

Estimation of summation growth II

- Analogously, if $f(x)$ is a decreasing function

$$\boxed{\int_0^N f(x)dx \geq S_N \geq \int_1^{N+1} f(x)dx}$$

- Some summations that can be estimated with this method:

$$S_N = \sum_{i=1}^N i^k \rightarrow S_N \sim \frac{N^{k+1}}{k+1} \quad \text{or also} \quad S_N = \frac{N^{k+1}}{k+1} + O(N^k)$$

$$S_N = \sum_{i=1}^N \log(i) = \log(N!) \rightarrow S_N \sim N \log(N)$$

$$H_N = \sum_{i=1}^N \frac{1}{i} \rightarrow H_N \sim \log(N) \quad (N\text{-th harmonic number})$$

Nota: We recommend to follow in the classroom or in the course notes the method by which these expressions are derived, since each of them have peculiarities regarding the estimation of the summation growth using integrals.

In this section we have learnt ...

- The **different types** of asymptotic comparisons between functions.
- How to **estimate the growth of several functions** with a closed form or using integral approximations.

Tools and techniques to work on...

- Basic work on estimations o , O , etc.
- Applications of estimating o , O , etc.
- Estimation of summation asymptotic growth using integrals.
- Exercises to solve: those recommended in section 2 in the exercise and problem sheets (at least !!!)



1.3 Algorithm complexity

Algorithm complexity

- So far, in our examples, the running time taken by an algorithm depended on each particular input : $n_A(I) \leq f_A(\tau(I))$.
- In some algorithms, there is a large difference in the ART depending on the specific input, e.g.,

$$N_{\text{BSort_v2}}([N, N-1, \dots, 1], 1, N) = 1/2(N^2 - N) \quad (\text{that's a lot !})$$

$$N_{\text{BSort_v2}}([1, 2, \dots, N], 1, N) = (N-1) \quad (\text{just a little bit})$$

- We would like to make our analysis a little bit more precise in terms of this observation.
- Thus, we define the **input space** of size **N** of an algorithm **A** as:
$$S_A(N) = \{I \text{ input of } A / \tau(I) = N\}$$

Example: BSort

- $S_{\text{BSort}}(N) = \{ (T, F, L) : L - F + 1 = N \} \dots ???$
- With any F , L y T this space is untreatable (too large)
 \Rightarrow we need to restrict this space.
- One possible simplification is to consider $F=1$ and $L=N$.
- Another possible simplification is to consider $T \in \Sigma_N$ (permutations of size N), since the important thing is the disorder among the elements of the array.
- With this simplifications

$$S_{\text{BSort}}(N) = \Sigma_N \quad \text{and} \quad |S_{\text{BSort}}(N)| = N!$$

Example: BSearch

- $S_{\text{BSearch}}(N) = \{ (T, F, L, k) : \text{any } k, L - F + 1 = N, T \text{ sorted} \}$
- A reasonable simplification is considering $F=1$ y $L=N$.
- Since T is sorted, we can also consider $T = [1 \dots N]$
- For keys in the table we consider $k=1, \dots, N$
- BSearch performs the same key comparisons for any k not in $T \Rightarrow$ we add the error key '**other**'
- With these simplifications:
$$S_{\text{BSearch}}(N) = \{ 1, 2, \dots, N, \text{other} \} \quad \text{and}$$
$$|S_{\text{BSearch}}(N)| = N + 1$$

Worst, best and average cases

Complexity definitions

Worst case

$$W_A(N) = \max \{n_A(I) / I \in S_A(N)\}$$

Best case

$$B_A(N) = \min \{n_A(I) / I \in S_A(N)\}$$

Average case

$$A_A(N) = \sum_{I \in S_A(N)} n_A(I) p(I)$$

where $p(I)$ is the probability of input I

Note that $B_A(N) \leq A_A(N) \leq W_A(N)$

Complexity in the worst case

- How to estimate the worst case of an algorithm?

- Step 1: Find $f_A(N)$ such that when $\tau(I)=N$ then

$$n_A(I) \leq f_A(N)$$

- Step 2: Find an input \hat{I} that corresponds to the worst input to the algorithm in the sense that

$$n_A(\hat{I}) \geq f_A(N)$$

- Because of step 1, $W_A(N) \leq f_A(N)$

- Because of step 2, $W_A(N) \geq f_A(N)$

- Thus $W_A(N) = f_A(N)$

- The best case can be estimated similarly.

- Exercise: write step 1 and 2 for the best case.

BSort worst case

- Example: $W_{\text{BSort2}}(N)$

- (1) We already saw that for all $\sigma \in \Sigma_N$
$$n_{\text{BSort2}}(\sigma) \leq N^2/2 + O(N)$$

- (2) When $\sigma = [N, N-1, N-2, \dots, 1]$ then
$$n_{\text{BSort2}}(\sigma) = N^2/2 + O(N)$$

- Considering (1) and (2)

$$W_{\text{BSort2}}(N) = N^2/2 + O(N)$$

- Exercise: show that $B_{\text{BSort2}}(N) = N-1$

Complexity in the average case

- The average case is usually the most difficult one to calculate.
- In general (but not always) we can assume equiprobability in the input space, i.e.,

$$p(I)=1/|S_A(N)|$$

- Examples:
 - In BSort $p(\sigma)=1/N!$, and
 - In BSearch $p(k)=1/(N+1)$

Average case for linear search

- We consider first equiprobability for the linear search

```
LSearch(array T, ind F, ind L, key k)
    for i=F to L:
        if T[i]==k:
            return k;
    return ERROR;
```

- Basic operation: key comparison (if $T[i]==k$)
- $S_{\text{LSearch}}(N) = \{1, 2, \dots, N, \text{other}\}$

Average case for linear search II

- $S_{\text{LSearch}}(N) = \{1, 2, 3, \dots, N, \text{other}\}, |E_{\text{LSearch}}(N)| = N+1$
- Then,
 - $p(k=i) = 1/(N+1) \quad (1 \leq i \leq N)$
 $p(k=\text{other}) = 1/(N+1)$
 - If $k \neq T[i] \Rightarrow n_{\text{LSearch}}(k) = N$
If $k = T[i] \Rightarrow n_{\text{LSearch}}(k) = i$
- Thus,
 - $W_{\text{LSearch}}(N) = N,$
 - $B_{\text{LSearch}}(N) = 1.$
 - $A_{\text{LSearch}}(N) = N/2 + O(1)$

Average case for linear search III

- Example: Successful search with non-equiprobable LSearch:

- In this case we have $p(k == T[i]) = \frac{1}{C_N} f(i)$
- C_N is a normalization constant that guarantees that

$$\sum_{i=1}^N \frac{1}{C_N} f(i) = 1, \text{ i.e., } C_N = \sum_{i=1}^N f(i)$$

$$\begin{aligned} A_{LSearch}(N) &= \sum_{i=1}^N n_{LSearch}(k = T[i]) p(k == T[i]) = \sum_{i=1}^N i \frac{1}{C_N} f(i) = \\ &= \frac{1}{C_N} \sum_{i=1}^N i f(i) = \frac{S_N}{C_N}, \text{ where } S_N = \sum_{i=1}^N i f(i) \end{aligned}$$

- S_N and C_N can be approximated (e.g., by integrals)
- We finally obtain an approximation for A.

Average case for linear search IV

- For example, if we have $p(k == T[i]) = \frac{1}{C_N} \frac{\log(i)}{i}$

$$\text{then } C_N = \sum_{i=1}^N \frac{\log(i)}{i} \quad \text{and} \quad S_N = \sum_{i=1}^N i \frac{\log(i)}{i} = \sum_{i=1}^N \log(i)$$

- Approximating by integrals we have

$$C_N \sim \frac{1}{2} (\log(N))^2 \quad \text{and} \quad S_N \sim N \log(N)$$

- We can easily see that:

$$A_{BLin}(N) \sim \frac{2N}{\log(N)}$$

- This last step is not difficult but we have to demonstrate the last asymptotic equivalence.

In this section we have learnt...

- The expressions for the worst, best and average cases of an algorithm.
- How to calculate the worst, best and average cases for some simple (BSort y LSearch).

Tools and techniques to work on...

- Calculation of worst and best cases in simple algorithms
- Calculation of the average case of linear searches and other variants.
- Exercises to solve: those recommended in section 3 of the exercise and problem sheets (at least!!!).