Sorting and Complexity Analysis

Algorithmic Efficiency

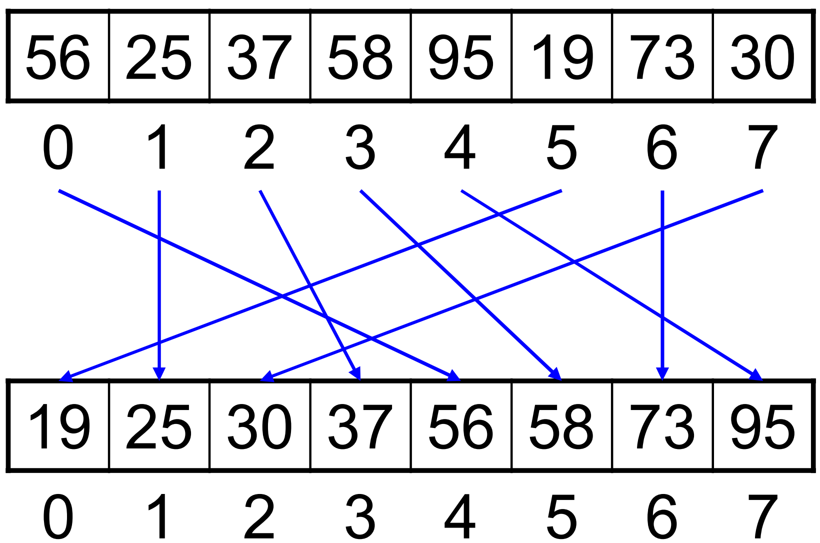
- We sometimes say "algorithm *A* is *faster or more efficient* than algorithm *B*." But how do we actually measure efficiency?

- We shall discuss how we can qualitatively measure the efficiency of an algorithm.

- We start with the problem of ***sorting***.

The Sorting Problem

- The problem of ***sorting*** is to *reorder* the elements in an array so that they fall in some defined sequence.



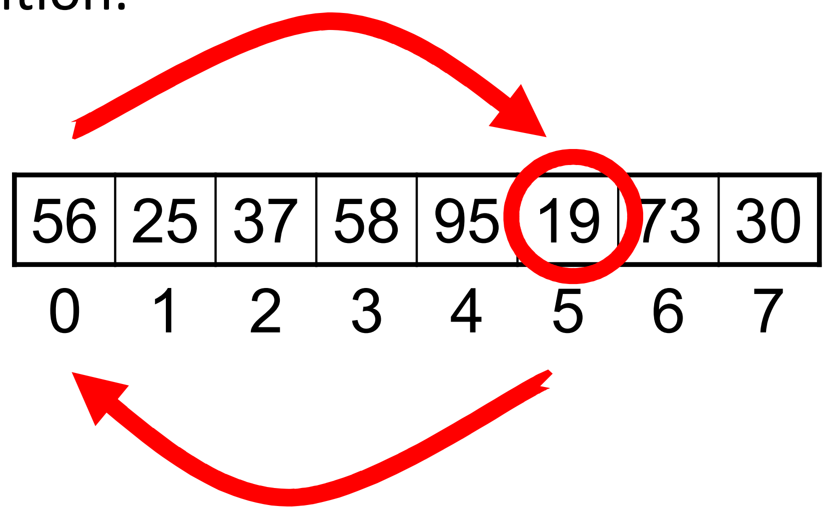
Ascending order

The Selection Sort Algorithm

- One of the simplest sorting algorithms is the ***selection sort***.

- The algorithm goes through each array *position* and *selects* a suitable value for that position.

Position 0



array[0] : swap with the smallest value of array[0] to array[n]

array[1] : swap with the smallest value of array[1] to array[n]

array[2] : swap with the smallest value of array[2] to array[n]

:

:

The Selection Sort Algorithm

Initial:	56	25	37	58	95	19	73	30
Round 0:	**19**	25	37	58	95	56	73	30
Round 1:	19	**25**	37	58	95	56	73	30
Round 2:	19	25	**30**	58	95	56	73	37
Round 3:	19	25	30	**37**	95	56	73	58
Round 4:	19	25	30	37	**56**	95	73	58
Round 5:	19	25	30	37	56	**58**	73	95
Round 6:	19	25	30	37	56	58	**73**	95

selects a value for:

- `array[0]` 56 : swap with 19
- `array[1]` 25 : no swap
- `array[2]` 37 : swap with 30
- `array[3]` 58 : swap with 37
- `array[4]` 95 : swap with 56
- `array[5]` 95 : swap with 58
- `array[6]` 73 : no swap

Goes through each array *position* and *selects* a suitable value for that position.

The Selection Sort Algorithm

- Round 0: find the smallest element in `array[0 … n-1]` and exchange it with `array[0]`.
- Round 1: find the smallest element in `array[1 … n-1]` and exchange it with `array[1]`.
- Round 2: find the smallest element in `array[2 … n-1]` and exchange it with `array[2]`.
- …
- Round `i`: find the smallest element in `array[i … n-1]` and exchange it with `array[i]`.
- …
- Round `n-2`: find the smallest element in `array[n-2 … n-1]` and exchange it with `array[n-2]`.

Selection Sort Implementation

```
void SelectionSort(int array[], int n) {
    int i, j, k;
    for (i = 0; i < n - 1; i++) {

        (Find the index k such that array[k] is
         the smallest in array[i … n-1].)


        (Exchange array[i] and array[k].)

    }
}
```

Round i

Round i: find the smallest element in array[i … n-1] and exchange it with array[i].

Round 0 to Round n-2

Selection Sort Implementation

```
void SelectionSort(int array[], int n) {
   int i, j, k;
   for (i = 0; i < n - 1; i++) {
      k = i;
      for (j = i + 1; j < n; j++)
         if (array[j] < array[k])
            k = j;

      (Exchange array[i] and array[k])

   }
}
```

Round `i`

Find the index `k` such that `array[k]` is the smallest in `array[i … n-1]`

Round `i`: find the smallest element in `array[i … n-1]` and exchange it with `array[i]`.

Selection Sort Implementation

```
void SelectionSort(int array[], int n) {
   int i, j, k, tmp;
   for (i = 0; i < n - 1; i++) {
      k = i;
      for (j = i + 1; j < n; j++)
         if (array[j] < array[k])
            k = j;
      tmp = array[i];
      array[i] = array[k];
      array[k] = tmp;
   }
}
```

Round `i`

Find the index `k` such that `array[k]` is the smallest in `array[i … n-1]`

Exchange `array[i]` and `array[k]`

Round `i`: find the smallest element in `array[i … n-1]` and exchange it with `array[i]`.

How Efficient is Selection Sort?

- Some *empirical* measurements

N	Time (s)
10,000	0.265
20,000	1.061
40,000	4.260
100,000	26.797
110,000	32.557
120,000	38.721
140,000	54.142
200,000	110.032

Observations:

Doubling N increases the running time by 4 times roughly.

Multiplying N by 10 times increases the running time by 100 times roughly.

CPU: AMD Athlon™ 64 3500+ (2.2GHz)

Analyzing Selection Sort

Initial:	56	25	37	58	95	19	73	30

- Round 0: must consider all N elements in `array[0 … n-1]`

Round 0:	19	25	37	58	95	56	73	30

- Round 1: must consider N **- 1** elements in `array[1 … n-1]`

Round 1:	19	25	37	58	95	56	73	30

- Round 2: must consider N **- 2** elements in `array[2 … n-1]`

Round 2:	19	25	30	58	95	56	73	37

- Round 3: must consider N **- 3** elements…

Round `i`: find the smallest element in `array[i … n-1]` and exchange it with `array[i]`.

Analyzing Selection Sort

- The total running time is roughly *proportional* to

$$N + (N\text{-}1) + (N\text{-}2) + \ldots + 3 + 2 + 1$$

$$= N(N+1)/2$$

$$= (N^2 + N)/2$$

How Big is $(N^2 + N)/2$?

N	Time (s)	$F(N)=(N^2 + N)/2$
10,000	0.265	50,005,000
20,000	1.061	200,010,000
40,000	4.260	800,020,000
100,000	26.797	5,000,050,000
110,000	32.557	6,050,055,000
120,000	38.721	7,200,060,000
140,000	54.142	9,800,070,000
200,000	110.032	20,000,100,000

Observations:

Doubling N increases $F(N)$ by 4 times roughly.

Multiplying N by 10 times increases $F(N)$ by 100 times roughly.

Analyzing an Algorithm

- Precise running time of an algorithm depends on specific computer hardware.

- The *essence* of analyzing the selection sort, however, is *how the algorithm responds to changes in the size N of the array*.
  - That is,

Doubling N increases the running time by 4 times roughly.

Computational Complexity

- The relationship between the problem size N and the performance of an algorithm as N becomes large is called the ***computational complexity*** (or time complexity) of the algorithm.

- To denote computational complexity, we use the ***big-O notation***.

Big-O Notation

- The ***big-O notation*** is used to provide a *quantitative insight* as to how changes in the problem size N affect the algorithmic performance as N becomes *large*.

- For example, as we shall see, the computational complexity of selection sort is $O(N^2)$ (Read as "big-O of N squared.")

Standard Simplifications of Big-O

- Before giving the formal definition, let's see how we can simplify a formula when using big-O notation.

- We illustrate the simplifications using the formula obtained from selection sort:

$$(N^2 + N)/2$$

Simplification Rule 1

- *Eliminate any term whose contribution to the total becomes* ***in****significant as N becomes large.*

- Example: $(N^2 + N)/2$

  $= N^2/2 + N/2$

  $= O(N^2/2)$

Simplification Rule 1

N	$N^2/2$	$N/2$	$(N^2 + N)/2$
10	50	5	55
100	5,000	50	5,050
1,000	500,000	500	500,500
10,000	50,000,000	5,000	50,005,000
100,000	5,000,000,000	50,000	5,000,050,000

The term $N/2$ (comparing with $N^2/2$) becomes ***in***significant to the total value when N becomes large.

Simplification Rule 1

- When a formula involves a summation of several terms, the *fastest growing term* alone will *control* the running time of the algorithm for large N.

- More examples
  - $N + 1 = O(N)$
  - $N^3 + 1000N^2 + N = O(N^3)$

Simplification Rule 2

- *Eliminate any **constant factors**.*

- Example: $(N^2 + N)/2$

  $= O(N^2/2)$ Rule 1

  $= O(N^2)$ Rule 2

Simplification Rule 2

Increase by100 times
when N is increased by 10 times

N	$N^2/2$	N^2
10	50	100
100	5,000	10,000
1,000	500,000	1,000,000
10,000	50,000,000	100,000,000
100,000	5,000,000,000	10,000,000,000

The constant factor 1/2 has ***no*** effect on the growth rate.

Simplification Rule 2

- What we want to capture in computational complexity is how changes in N affect the algorithmic performance.
- Constant factors have no effect on the growth rate.

- More examples
  - $10000N^{0.5} = O(N^{0.5})$
  - $0.0001N^3 + 10000N^2 + N + 3 = O(N^3)$

Exercises

- $2N^9 + N = O(?)$
- $7N - 2N^{1/2} + 4 = O(?)$
- $N^{3/2} - 2N^{1/2} = O(?)$
- $2N + 4\log N = O(?)$
- $N^2 + N\log N = O(?)$

Implications of Computational Complexity

- Recall that the computational complexity of selection sort is $O(N^2)$.
- An implication on $O(N^2)$ is that the *running time grows by the square of the increase in the problem size*.
- This *precisely* captures the performance of selection sort, which is

doubling N increases the running time by 4 times,

multiplying N by 10 times increases the running time by 100 times

Determining Computational Complexity from Code Structure

- What is the computational complexity of the following function?

```
double Average(double *array, int n) {
   int i;
   double total = 0.0;
   for (i = 0; i < n; i++)
      total += array[i];
   return total / n;
}
```

Loop:
N iterations

Each other statement executed *once*.

Determining Computational Complexity from Code Structure

- What is the computational complexity of the following function?

```
double Average(double *array, int n) {
   int i;
   double total = 0.0;
   for (i = 0; i < n; i++)
      total += array[i];
   return total / n;
}
```

Each of these statements runs in ***constant time*** (independent to N).

Constant is denoted as $O(1)$ in big-O notation.

- Hence, computational complexity is $O(N)$.
- Commonly called ***linear time***.

Determining Computational Complexity from Code Structure

- In general, we can determine the time complexity simply by finding the piece of the code that is executed *most often*.

```
for (i = 0; i < n; i++)
    total += array[i];
```

- However, if an expression or statement involves *function calls*, it must be accounted *separately*.

Determining Computational Complexity from Code Structure

- What about this one?

```
double Variance(double array[], int n) {
   double k, *temp;
   int i;
   temp = (double *)malloc(n * sizeof(double));
   for (i = 0; i < n; i++) {
      k = array[i] - Average(array, n);
      temp[i] = k * k;
   }
   return Average(temp, n);
}
```

System dependent, usually $O(1)$ or $O(N)$ time.

Loop: N iterations

$O(N)$ time

$O(1)$ time

Totally $O(N(N+1)) = O(N^2)$ time for this part.

$O(N)$ time

- $O(N^2 + N) = O(N^2)$
- Commonly called ***quadratic time***.

Determining Complexity from Code Structure

- With a little bit revision,

```
double Variance(double array[], int n) {
    double k, mean, *temp;
    int i;
    temp = (double *)malloc(n * sizeof(double));
    mean = Average(array, n);
    for (i = 0; i < n; i++) {
        k = array[i] - mean;
        temp[i] = k * k;
    }
    return Average(temp, n);
}
```

$O(N)$ time

Loop body: N iterations

$O(1)$ time

$O(N)$ time

Totally $O(N \times 1) = O(N)$ time only for this part.

- it improves to $O(N + N) = O(N)$.

Selection Sort Revisited

```
void SelectionSort(int array[], int n) {
    int i, j, k;
    for (i = 0; i < n - 1; i++) {
        k = i;
        for (j = i + 1; j < n; j++)
            if (array[j] < array[k])
                k = j;
        j = array[i];
        array[i] = array[k];
        array[k] = j;
    }
}
```

Outer loop: $O(N)$ iterations

Inner loop: $O(N)$ iterations

Each *single* statement/expression executes in $O(1)$ time.

- $O(N \times N) = O(N^2)$

Formal Definition of Big-O

- Definition: $T(N) = O(f(N))$ if and only if
  - there are positive constants n_0 and c such that for every value of $N \geq n_0$, the following condition holds:

    $T(N) \leq c \times f(N)$

- As long as N is "*large enough*," $T(N)$ is always bounded by a constant multiple of $f(N)$.

for $N \geq n_0$, $T(N) \leq c \times f(N)$

Example: why $(N^2 + N)/2 = O(N^2)$?

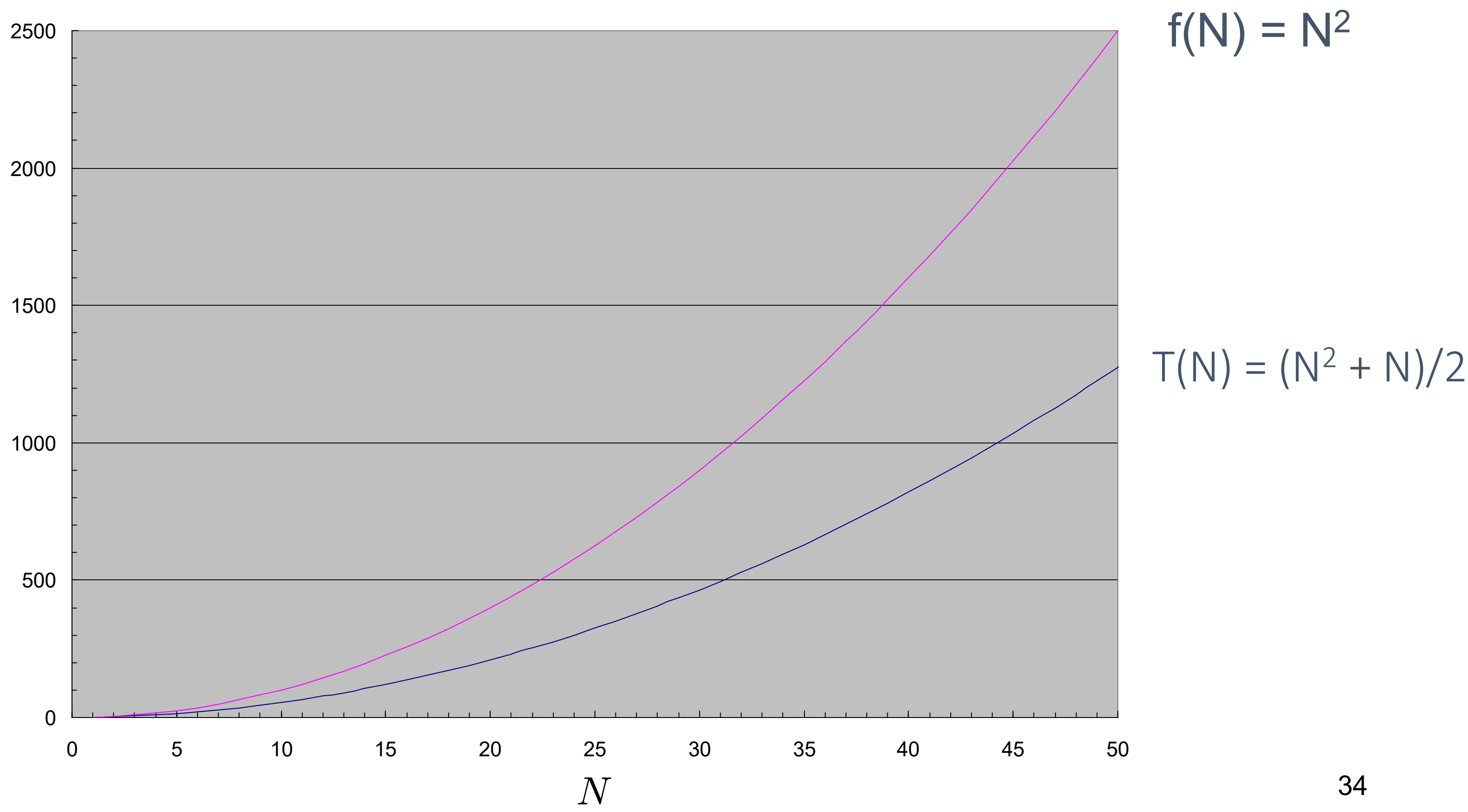
- To prove $(N^2 + N)/2 = O(N^2)$, we need to find constants n_0 and c so that for all values of $N \geq n_0$, $(N^2 + N)/2 \leq cN^2$

- We know that $N \leq N^2$ when $N \geq 1$
- Therefore, for all $N \geq n_0 = 1$, we have

$$
\begin{aligned}
(N^2 + N)/2 &\leq (N^2 + N^2)/2 \\
&= N^2 \\
&= 1N^2
\end{aligned}
$$

- Thus, setting $n_0 = 1$ and $c = 1$ completes the proof

$(N^2 + N)/2 = O(N^2)$



2500
2000
1500
1000
500
0
0
5
10
15
20
25
30
35
40
45
50
N
f(N) = N2
T(N) = (N2 + N)/2

Examples

(A) $2N + 4 = O(N)$

for all $N \geq 4$, $2N + 4 \leq 2N + N = 3N$

(n_0 = 4 and c = 3)

(B) $N^8 + 1000N^3 = O(N^8)$

for all $N \geq 4$, $N^8 + 1000N^3 \leq N^8 + N^5 N^3 = 2N^8$

Note that
when N = 3, $N^5 = 243 < 1000$
when N = 4, $N^5 = 1024 > 1000$ (or $1000 < N^5$)

(n_0 = 4 and c = 2)

Polynomials

- In general, given a polynomial $P(N)$ of degree k,

$$P(N) = a_k N^k + a_{k-1} N^{k-1} + \ldots + a_2 N^2 + a_1 N + a_0$$

where a_0, …, a_k and k are constants, we can prove that

$$P(N) = O(N^k)$$

Examples

$$4N + \log N = O(N)$$

for all $N \geq 1$,
$$\begin{aligned} 4N + \log N &\leq 4N + N \\ &= 5N \end{aligned}$$

(n_0 = 1 and c = 5)