COS 460 – Algorithms (week 03)

Analysis of algorithms.

Comparing algorithms and predicting performance.

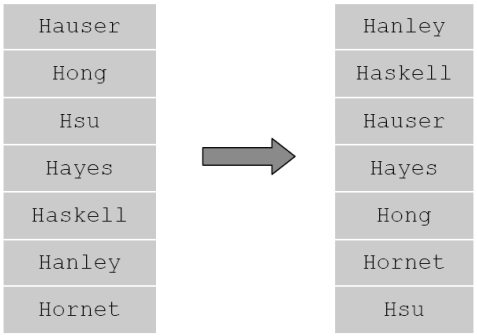
Scientific method:

- Observe some feature of the universe.
- Hypothesize a model that is consistent with observation.
- Predict events using the hypothesis.
- Verify the predictions by making further observations.
- Validate the theory by repeating the previous steps until the hypothesis agrees with the observations.

Running Time. Case Study: Sorting

Sorting problem:

- **Given N items, rearrange them in ascending order.**
- **Applications: statistics, databases, data compression, computational biology, computer graphics, scientific computing, ...**



Insertion sort.

- Brute-force sorting solution.
- Move left-to-right through array.
- Exchange next element with larger elements to its left, one-by-one.

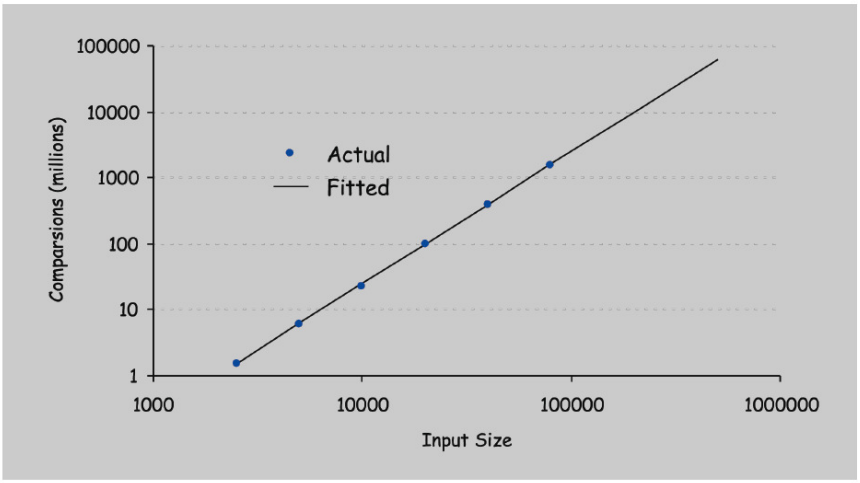
```
void insertionSort(int N, double a[])
 {
  for (int i = 0; i < N; i++)
   for (int j = i; j > 0; j--)
     if (a[j] < a[j-1]) swap(a[j], a[j-1]);
     else break;

  }
```

Observe and tabulate running time for various values of N.

- Data source: N random numbers between 0 and 1.

N	Comparisons
5,000	6.2 million
10,000	25 million
20,000	99 million
40,000	400 million
80,000	16 million

Insertion Sort: Experimental Hypothesis
Data analysis. Plot # comparisons vs. input size on log-log scale.



Regression. Fit line through data points = aN^b.
Hypothesis. # comparisons grows quadratically with input size ! $N^2/4$.

Experimental hypothesis.

- Measure running times, plot, and fit curve.
- Model useful for predicting, but not for explaining

Theoretical hypothesis.

- Analyze algorithm to estimate # comparisons as a function of:
  - number of elements N to sort
  - average or worst case input
- Model useful for predicting and explaining.
- Model is independent of a particular machine or compiler.

Difference. Theoretical model can apply to machines not yet built.

Worst case. [descending]

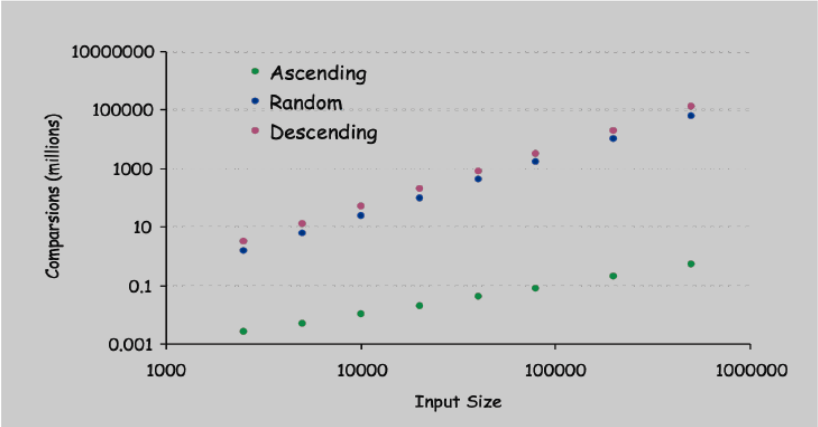
- Iteration i requires i comparisons.
- Total = 0 + 1 + 2 + … + N-2 + N-1 = N (N-1) / 2.

Average case. [random]

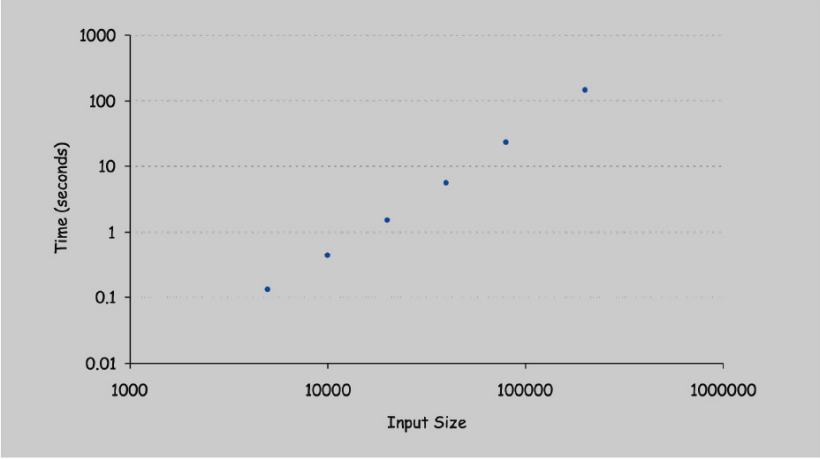
- Iteration i requires i/2 comparisons on average.
- Total = 0 + 1/2 + 2/2 + … + (N-1)/2 = N (N-1) / 4.

Validation. Theory agrees with observations.

Analysis	Input	Comparisons
Worst	Descending	$N^2 / 2$
Average	Random	$N^2 / 4$
Best	Ascending	N



Data source: N random numbers between 0 and 1.



Timing in C++: Measure time between beginning and end of computation.

`clock()` returns the number of clock ticks elapsed since the program was launched.
The macro constant expression `CLOCKS_PER_SEC` specifies the relation between a clock tick and a second (clock ticks per second).

```
#include<iostream>
#include<ctime>
using namespace std;

int main()
{
 clock_t t0 = clock();
 // example run:
 for(long long int i=0;i<1e9;i++);
 cout <<  ((double) (clock() - t0))/CLOCKS_PER_SEC << endl;
 system("pause");
}
```

Factors that affect running time

- Machine.
- Compiler.
- Algorithm.
- Input data.

More factors

- Caching.
- Garbage collection.
- CPU used by other processes.

How Formulate a Hypothesis about Running Time

Total running time: sum of cost × frequency for all of the basic ops.

- Cost depends on machine, compiler.
- Frequency depends on algorithm, input.

Cost for sorting.

- A = # exchanges.
- B = # comparisons.
- Cost on a typical machine = 11A + 4B.

Frequency of sorting ops.

- N = # elements to sort.
- Selection sort: A = N-1, B = N(N-1)/2.

An easier alternative.

(i) Analyze asymptotic growth as a function of input size N.
(ii) For medium N, run and measure time.
(iii) For large N, use (i) and (ii) to predict time.

Asymptotic growth rates.

Estimate as a function of input size N.
– N, N log N, N^2, N^3, 2^N, N!
Ignore lower order terms and leading coefficients.
– Example: $6N^3 + 17N^2 + 56$ is asymptotically proportional to N^3

Seconds	Equivalent
1	1 second
10	10 seconds
10^2	1.7 minutes
10^3	17 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
. . .	forever
10^{17}	age of universe

Why It Matters

Run time in nanoseconds -->		1.3 N^3	10 N^2	47 N $\log_2 N$	48 N
Time to solve a problem of size	1000	1.3 seconds	10 msec	0.4 msec	0.048 msec
	10,000	22 minutes	1 second	6 msec	0.48 msec
	100,000	15 days	1.7 minutes	78 msec	4.8 msec
	million	41 years	2.8 hours	0.94 seconds	48 msec
	10 million	41 millennia	1.7 weeks	11 seconds	0.48 seconds
Max size problem solved in one	second	920	10,000	1 million	21 million
	minute	3,600	77,000	49 million	1.3 billion
	hour	14,000	600,000	2.4 trillion	76 trillion
	day	41,000	2.9 million	50 trillion	1,800 trillion
N multiplied by 10, time multiplied by		1,000	100	10+	10

Big O notation

Big O notation or Big Oh notation, and also Landau notation or asymptotic notation, is a mathematical notation used to describe the **asymptotic behavior** of functions. Its purpose is to characterize a function's behavior for *very large* (or *very small*) inputs in a simple but rigorous way that enables comparison to other functions. More precisely, the symbol O (Greek omicron) is used to describe an asymptotic upper bound for the magnitude of a function in terms of another, usually simpler, function. There are also other symbols o, Ω, ω, and Θ for various other upper, lower, and tight bounds.

It has a main area of application in computer science, where it is useful in the analysis of the complexity of algorithms.

For example, let us the time (or the number of steps) that an algorithm takes to complete a problem of size n be found to be $T(n) = 4n^2 - 2n + 2$.

As n grows large, the n^2 term will come to dominate, so that all other terms can be neglected — for instance when $n = 500$, the term $4n^2$ is 1000 times as large as the $2n$ term, and so ignoring the latter would have negligible effect on the expression's value for most purposes.

Further, the coefficients become irrelevant as well if we compare to any other order of expression, such as an expression containing a term n^3 or 2^n. Even if $T(n) = 1{,}000{,}000n^2$, if $U(n) = n^3$, the latter will always exceed the former once n grows larger than 1,000,000 ($T(1{,}000{,}000) = 1{,}000{,}000^3 = U(1{,}000{,}000)$).

So the big O notation captures what remains: we write $T(n) \in O(n^2)$ (or, $T(n) = O(n^2)$) and say that the algorithm has *order of* n^2 time complexity.

Formal definition:

Suppose $f(x)$ and $g(x)$ are two functions defined on some subset of the real numbers. We say

$f(x)$ is $O(g(x))$ as $x \to \infty$
if and only if there exists x_0 and $M > 0$ such that $|f(x)| < M\,|g(x)|$ for all $x > x_0$

Example:

Consider $f(x) = 6x^4 - 2x^3 + 5$ and $g(x) = x^4$. Show that $f(x)$ is $O(g(x))$.

We can prove that $|f(x)| < C|g(x)|$ for all $x > 1$, where C is a constant. Proof:

For $x > 1$:
$|6x^4 - 2x^3 + 5| \le 6x^4 + 2x^3 + 5$,
$|6x^4 - 2x^3 + 5| \le 6x^4 + 2x^4 + 5x^4$, because $x^4 > x^3$ and $x^4 > 1$.
It follows that $|6x^4 - 2x^3 + 5| \le 13|x^4|$, so C = 13.

Properties

Multiplication by a constant: $O(kg(n)) = O(g(n))$, $k \neq 0$.

If **f**(**n**) is a polynomial of degree **d**, then **f**(**n**) is $\mathbf{O}(\mathbf{n^d})$, i.e., drop lower-order terms and drop constant factors.
Use the smallest possible class of functions, i.e. say "2**n** is **O**(**n**)" instead of "2**n** is $\mathbf{O}(\mathbf{n^2})$"

Use the simplest expression of the class, i.e. say "3**n**+5 is **O**(**n**)" instead of "3**n**+5 is **O**(3**n**)"

Big-Oh and Growth Rate

The big-Oh notation gives an upper bound on the growth rate of a function
The statement "**f**(**n**) is **O**(**g**(**n**))" means that the growth rate of **f**(**n**) is no more than the growth rate of **g**(**n**)
We can use the big-Oh notation to rank functions according to their growth rate

Relatives of Big-Oh

Big-Omega: f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for $n \geq n_0$

Big-Theta: f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c'g(n) \leq f(n) \leq c''g(n)$ for $n \geq n_0$

Little-oh: f(n) is o(g(n)) if, for *any* constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \leq c\ g(n)$ for $n \geq n_0$

Little-omega: f(n) is $\omega(g(n))$ if, for *any* constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \geq c\ g(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation:
Big-Oh: f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)
Big-Omega: f(n) is $\Omega(g(n))$ if f(n) is asymptotically **greater than or equal** to g(n)
Big-Theta: f(n) is $\Theta(g(n))$ if f(n) is asymptotically **equal** to g(n)
Little-oh: f(n) is o(g(n)) if f(n) is asymptotically **strictly less** than g(n)
Little-omega: f(n) is $\omega(g(n))$ if is asymptotically **strictly greater** than g(n)

Running time is O(1) for elementary operations:

- Function call.
- Boolean operation.
- Arithmetic operation.
- Assignment statement.
- Access array element by index.

Logarithmic time. Running time is O(log N).
Searching in a sorted list. Given a sorted array of items, find index of query item.
O(log N) solution. Binary search.

```
int binarySearch(int size, string a[], string key)
{
  int left = 0;
  int right = size - 1;
  while (left <= right)
  {
    int mid = (left + right) / 2;
    int cmp = key.compare(a[mid]);
    if (cmp < 0) right = mid - 1;
    else if (cmp > 0) left = mid + 1;
    else return mid;
  }
  return -1;
}
```

Linear time. Running time is O(N).
Find the maximum. Find the maximum value of N items in an array.

```
double max = a[0];
for (int i = 1; i < N; i++)
{
 if (a[i] > max) max = a[i];
}
```

Linearithmic time. Running time is O(N log N).
Sorting. Given an array of N elements, rearrange in ascending order.
O(N log N) solution. **Mergesort**.

Quadratic time. Running time is $O(N^2)$.
Closest pair of points. Given N points in the plane, find closest pair.
$O(N^2)$ solution. Enumerate all pairs of points. (Efficient solution ?)

```
double min = MAXVALUE;
for (int i = 0; i < N; i++)
for (int j = i+1; j < N; j++)
{
  double dx = (x[i] - x[j]);
  double dy = (y[i] - y[j]);
  if (dx*dx + dy*dy < min) min = dx*dx + dy*dy;
}
```

Exponential time. Running time is $O(a^N)$ for some constant a > 1.
Finbonacci sequence: 1 1 2 3 5 8 13 21 34 55 …
$O(a^N)$ solution, where a is:

$$\frac{1}{2}\left(1+\sqrt{5}\right) = 1.618034....$$

Spectacularly inefficient!

```
int F(int N)
{
 if (n == 0 || n == 1) return n;
 else return F(n-1) + F(n-2);
}
```

Summary of Common Hypotheses

Complexity	Description	When N doubles, running time
1	Constant algorithm is independent of input size.	does not change
log N	Logarithmic algorithm gets slightly slower as N grows.	increases by a constant
N	Linear algorithm is optimal if you need to process N inputs.	doubles
N log N	Linearithmic algorithm scales to huge problems.	slightly more than doubles
N^2	Quadratic algorithm practical for use only on relatively small problems.	quadruples
2^N	Exponential algorithm is not usually practical.	squares!

RECURRENCES

Recurrences and Recursive Code

Many (perhaps most) recursive algorithms fall into one of two categories: tail recursion and divide-and-conquer recursion. We would like to develop some tools that allow us to fairly easily determine the efficiency of these types of algorithms.

Tail Recursion
Tail recursion is the kind of recursion that pulls off 1 "unit" from the "end" (tail) of the data and processes the remainder recursively:

```
foo (data, n)
{ \\ data is size n
  tailData = preProcess(data, n); \\ generate tailData with size n-1
  foo (tailData, n-1);            \\ process tailData
  postProcess(data, tailData, n); \\ put things together
}
```

If p and q are the running times of `preProcess()` and `postProcess()`, and f is the running time of `foo()`, then we have

$f(n) = p(n) + f(n-1) + q(n)$

This simplifies to $f(n) = f(n-1) + g(n)$, where $g(n) = p(n) + q(n)$.

Divide-and-Conquer
Now consider a different kind of recursive function:

```
foo (data, n)
{
 // split data into k chunks
 dataChunks = preProcess(data, n); // dataChunks is an array
 for (i=0; i < k; i++)
  { // process each chunk
      foo(dataChunk[i], n/k);
  }
 postProcess(data, dataChunks, n);
}
```

If p and q are the running times of `preProcess()` and `postProcess()`, and f is the running time of foo(), then we have

$f(n) = p(n) + kf(n/k) + q(n)$
This simplifies to $f(n) = kf(n/k) + g(n)$

A *recurrence* is an equation or an inequality that describes a function in terms of its value on smaller inputs.

Example 1: (This recurrence arises for a recursive program that loops through the input to eliminate one item)

$T(n) = T(n-1)+n$, for $n>1$ with $T(1)=1$

"Telescopic method":
$T(n)=T(n-1)+n = T(n-2)+(n-1)+n = T(n-3)+(n-2)+(n-1)+n = \ldots =$
$= T(1)+2+3+\ldots+(n-2)+(n-1)+n = n(n-1)/2$

Example 2: (This recurrence arises for a recursive program that halve the input in each step – e.g. binary search)

$T(n) = T(n/2)+1$, for $n>1$ with $T(1)=0$ (assume that n/2 is an integer division)
We may also assume for now that $n=2^k$, i.e. $k = \lg n$

$T(2^k) = T(2^{k-1}) +1 = T(2^{k-2}) + 1 +1 = \ldots = T(2^0) + k = k$. Solution $T(n) = \lg n$

Example 3: (This recurrence arises for a recursive program that halve the input in each step, but perhaps must eliminate every item in the input)

$T(n) = T(n/2)+n$, for $n>1$ with $T(1)=0$ (assume that n/2 is an integer division)
We may also assume for now that $n=2^k$, i.e. $k = \lg n$

This telescopes to the sum $n + n/2 + n/4 + n/8 + \ldots + 4 + 2 + 1$.
By the formula of geometric series that evaluates to $2n – 1$.

Example 4: (This recurrence arises for a recursive program that has to make a linear pass through the input, before, during, or after it is split into two halves – most widely used in many standard divide-and-conquer algorithms – e.g. merge sort)

$T(n) = 2T(n/2)+n$, for $n>1$ with $T(1)=0$ (assume that n/2 is an integer division)
We may also assume for now that $n=2^k$, i.e. $k = \lg n$

Solution: $T(2^k) = 2T(2^{k-1}) + 2^k$

$T(2^k)/ 2^k = T(2^{k-1})/ 2^{k-1} + 1 = T(2^{k-2})/ 2^{k-2} + 1 + 1 = \ldots = k$; $T(n) = n \lg n$

Example 5:

$T(n) = 2T(n/2)+1$, for $n>1$ with $T(1)=0$
(assume that n/2 is an integer division)
We may assume for now that $n=2^k$, i.e. $k = \lg n$

Solution: $T(n) = 2n$.

Master Method

The master method provides a cookbook method for solving recurrences of the form

$T(n) = aT(n/b) + f(n)$

where $a \geq 1$ and $b > 1$ are constants and f(n) is an asymptotically positive function. The master method requires memorization of three cases but then the solution of many recurrences can be determined quite easily, often without pen and paper. The master method depends upon the master theorem.

Master Theorem. Let $a \geq 1$ and $b > 1$ be constants, let f(n) be a function, and let T(n) be defined as the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then T(n) can be bounded asymptotically as follows

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a - \varepsilon})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Example: Use the master method to give tight asymptotic bounds for the following recurrences.
a. $T(n) = 4T(n/2) + n$
b. $T(n) = 4T(n/2) + n^2$
c. $T(n) = 4T(n/2) + n^3$

Solution:
a. For the recurrence, $T(n) = 4T(n/2) + n$, we have $a = 4$, $b = 2$, $f(n) = n$, and thus $n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$. Since $f(n) = O(n^{\log_2 4 - \varepsilon})$, where $\varepsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.
b. For the recurrence, $T(n) = 4T(n/2) + n^2$, we have $a = 4$, $b = 2$, $f(n) = n^2$, and thus $n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$. Since $f(n) = \Theta(n^{\log_2 4})$, we can apply case 2 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2 \lg n)$.
c. For the recurrence, $T(n) = 4T(n/2) + n^3$, we have $a = 4$, $b = 2$, $f(n) = n^3$, and thus $n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$. Since $f(n) = \Omega(n^{\log_2 4 + \varepsilon})$, where $\varepsilon = 1/2$, case 3 of the master theorem applies if we can show the regularity condition holds for $f(n)$. For all n, $af(n/b) = 4(n/2)^3 = (1/2)(n/2)^3 \le (2/3)n^3 = cf(n)$ for $c = 2$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(f(n)) = \Theta(n^3)$.

Exercises:

1. Order the functions below by growth rate, i.e., order them in big-O terms. Prove your answers. Write and run simple code to demonstrate your answers:
a) n **b)** $n^{1/2}$ **c)** $\log n$ **d)** $\log \log n$ **e)** $(\log n)^2$ **f)** $n/\log n$ **g)** $n^{1/2}(\log n)^2$ **h)** $(3/2)^n$ **i)** $(1/3)^n$
2. Show that:
a) $n2 + 2n + 5$ is $O(n2)$
Hint: $(n2 + 2n + 5)/n2 = 1 + 2/n + 5/n2 \le 8$ for all positive integers n.
b) $(1.00001)^n$ is not $O(n^{1000})$
Hint: $\log(1.00001)^n = n \log 1.00001 = c\,n$, for some $c > 0$; $\log n^{1000} = 1000 \log n$;
$cn - 1000 \log n \to \infty$ as $n \to \infty$.
3. Do a worst-case running time analysis in big-O terms for the following function:

```
void f(int n)

{ for(int i=1, i <= n; i++)
  for(int j=1; j <= n; j++)
  { c[i][j] = 0;
    for(int k=1; k <= n; k++)
     c[i][j] += a[i][k]*b[k][j];
  }
}
```

4. Find an "efficient algorithm" for the following problems and then make a computer implementation:
(a) Given an array of integers, determine whether or not it is in increasing order.
Hint: Certainly, just loop across the array and check each element to see if it's larger than the preceding element – time complexity is definitely $O(n)$, where n is the length of the array.
(b) Given an array of integers, determine whether or not it can be partitioned into two arrays, each of which is in increasing order. For instance 3 1 5 2 4 can, but 4 8 1 5 3 can't.
Hint: The naive method of looking at all possible partitions and checking whether both parts are increasing is certainly of exponential complexity. However, an efficient method is still possible given the observation that if we have successfully partitioned an initial segment of the array, one of the parts must contain the maximum element seen so far. It is obviously in our best interest that the largest element of the other part be as small as possible. So, given the next element, if it is the maximum to this point add it to the "maximum containing part". If not, there is no choice but to add it to the other part if possible (i.e. if it is larger than the largest element of this part, but is not the current maximum). If this procedure fails then no partition is possible, and if it succeeds then we have demonstrated a partition. Since we do a fixed amount of work processing each element (check if it is larger than the previous maximum, check whether it is larger than the other tail element, update some records), the complexity is still $O(n)$.