



AUBURN
UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Efficiency

COMP 2210 – Dr. Hendrix

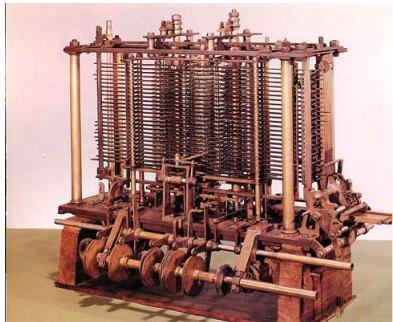
From the beginning



“As soon as an Analytic Engine exists, it will necessarily guide the future course of science. Whenever any result is sought by its aid, the question will arise – By what course of calculation can these results be arrived at by the machine in the shortest time?” – Charles Babbage (1864)



“In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selections amongst them for the purposes of a calculating engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.” – Ada Lovelace (1843)



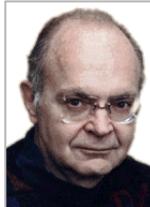
Computing the Numbers of Bernoulli on the Analytical Engine, by Ada Lovelace.

Caveat discipulus



“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity.”

William A. Wulf



“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.”

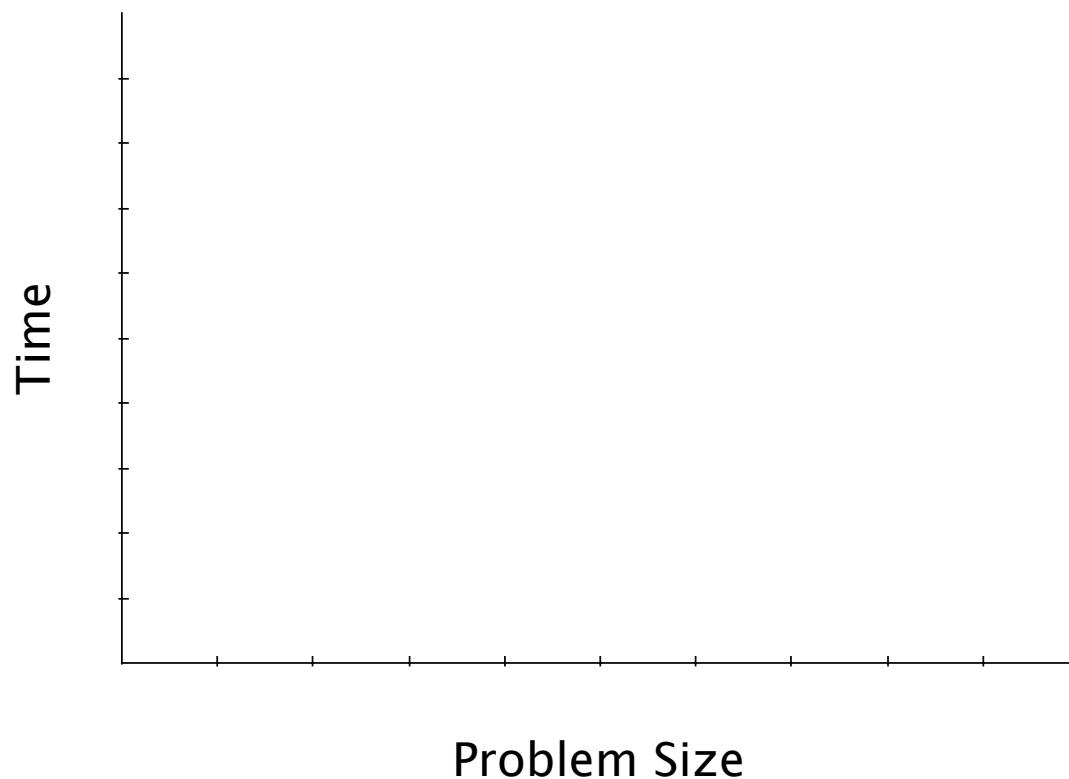
Donald E. Knuth



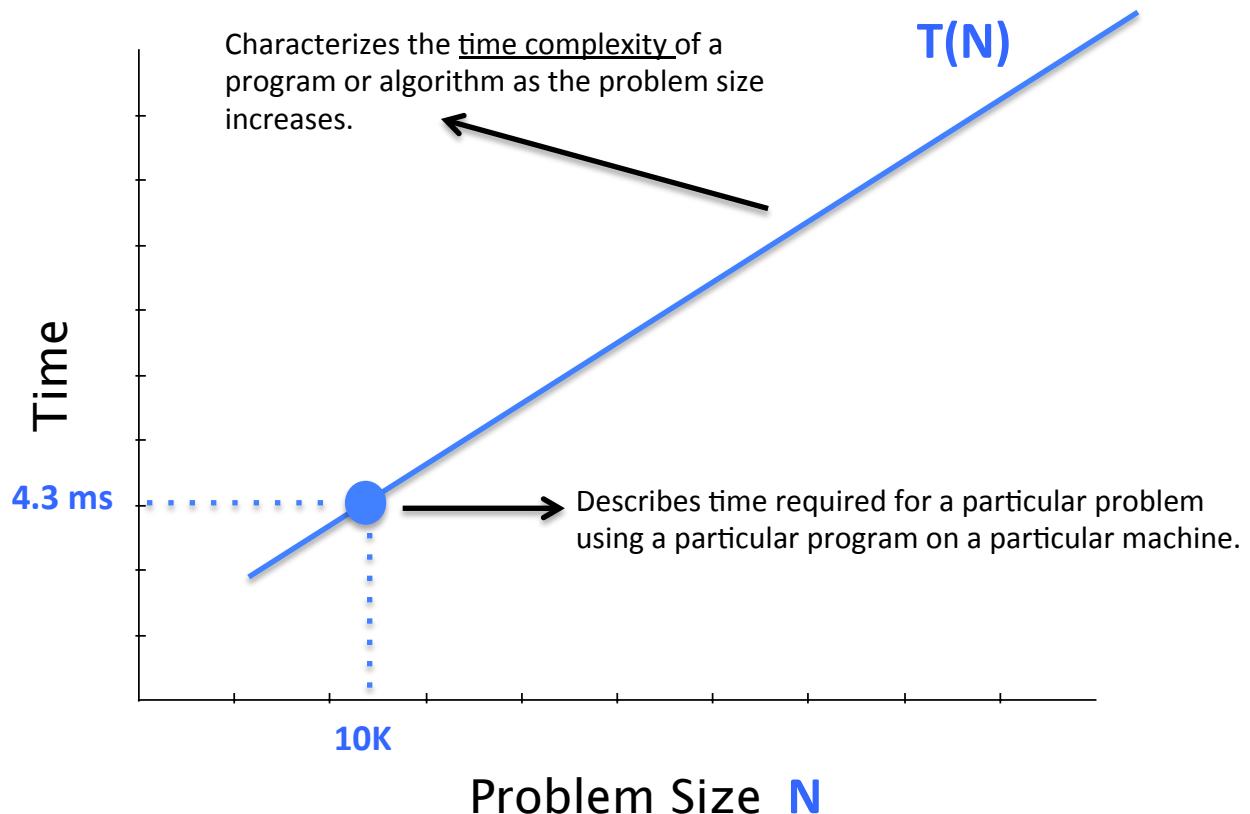
“We follow two rules in the matter of optimization: Rule 1: Don't do it. Rule 2 (for experts only): Don't do it yet - that is, not until you have a perfectly clear and unoptimized solution.”

Michael A. Jackson

Categorizing running time



Categorizing running time

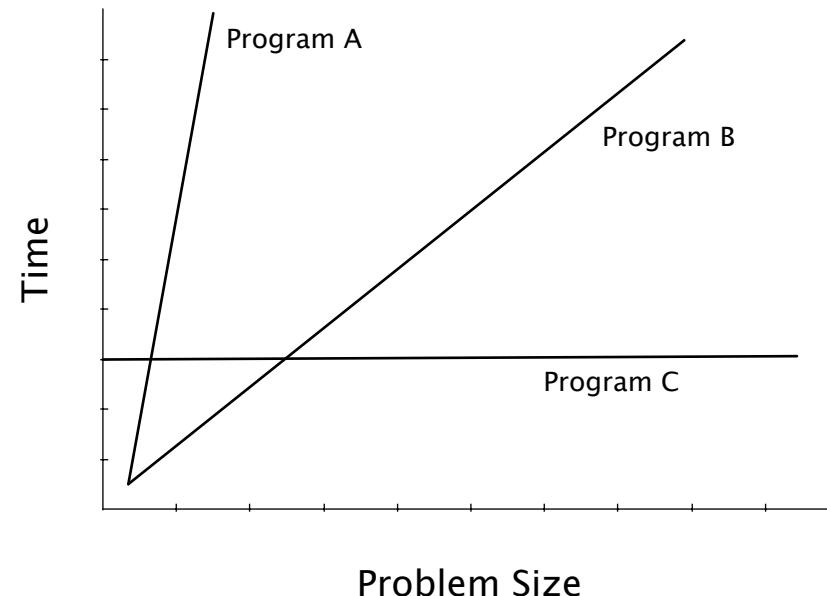


Participation



Q: Which program offers the best performance as the problem size scales up?

- A. Program A
- B. Program B
- C. Program C



Categorizing running time

The efficiency or time complexity of some programs may vary depending on the input, leading to different cases for the program's time complexity.

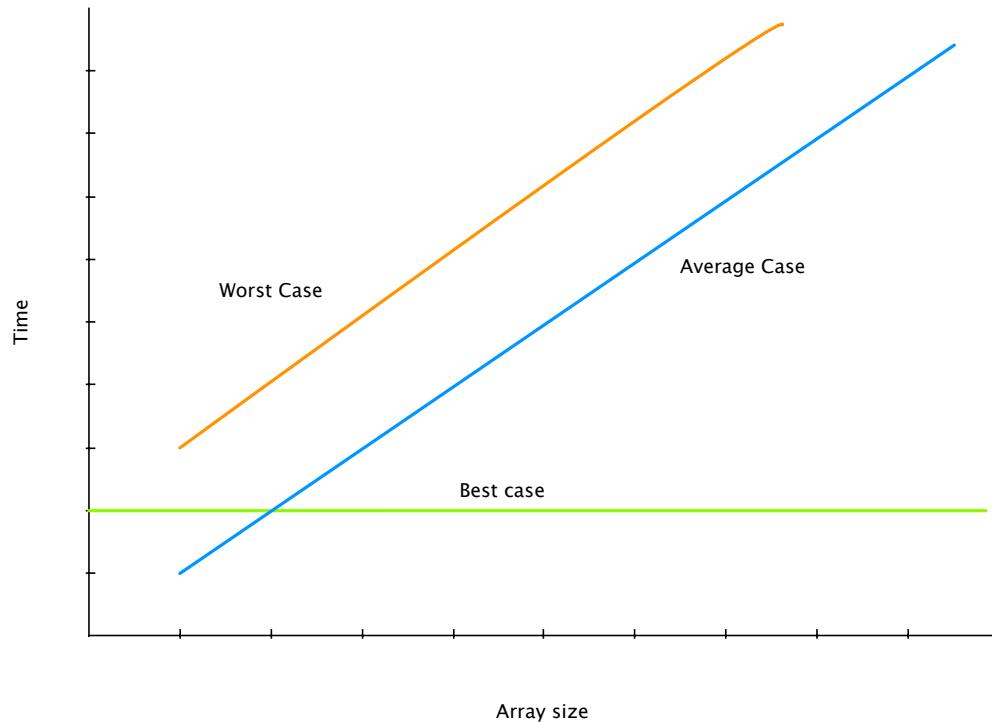
```
public int search(int[] a, int target) {  
    int i = 0;  
    while ((i < a.length) && (a[i] != target)) {  
        i++;  
    }  
    if (i < a.length)  
        return i;  
    else  
        return -1;  
}
```

```
int[] a = {2, 4, 6, 8, 10};  
  
search(a, 2);      Best case  
  
search(a, 6);      Average case  
  
search(a, 1);      Worst case
```

Categorizing running time

The efficiency or time complexity of some programs may vary depending on the input, leading to different cases for the program's time complexity.

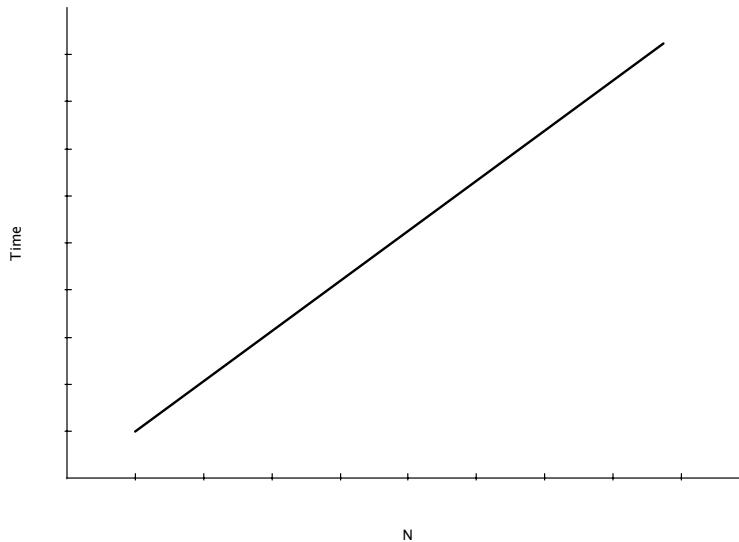
```
int[] a = {2, 4, 6, 8, 10};  
  
search(a, 2);      Best case  
  
search(a, 6);      Average case  
  
search(a, 1);      Worst case
```



Categorizing running time

Many algorithms are not sensitive to input so there are no different time complexity cases to consider.

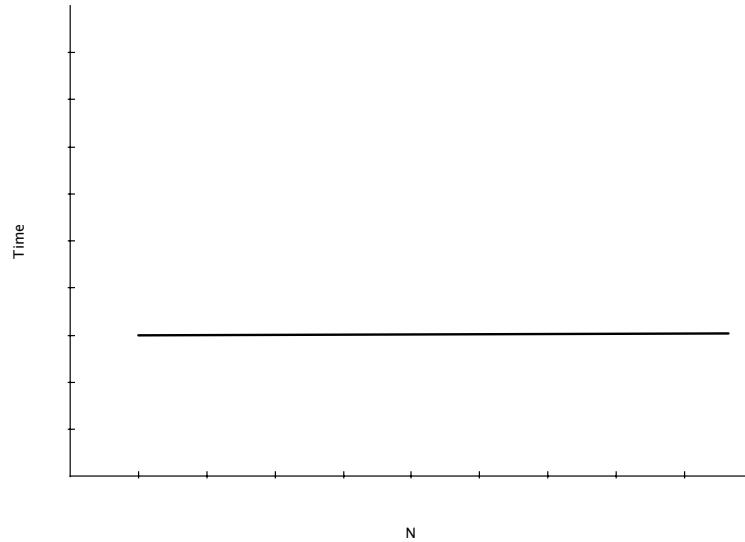
```
public int factorial(int N) {  
    int fact = 1;  
    for (int i = N; i > 1; i--) {  
        fact *= i;  
    }  
    return fact;  
}  
  
public static void main(String[] args) {  
    fact(5);  
    fact(10);  
    fact(15);  
}
```



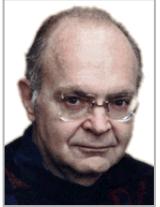
Categorizing running time

Many algorithms are not sensitive to input so there are no different time complexity cases to consider.

```
public int sigma(int N) {  
    return N * (N + 1) / 2;  
}  
  
public static void main(String[] args) {  
    sigma(5);  
    sigma(10);  
    sigma(15);  
}
```



Efficiency



"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

Donald E. Knuth

efficiency (noun) Skillfulness in avoiding wasted time and effort.

WordNet 3.1 <http://wordnetweb.princeton.edu/perl/webwn?s=efficiency>

So, one criteria that we can use to judge algorithms or programs is their skillfulness in avoiding wasted time and effort.

Efficiency

A boolean version of list search: return true if found, false if not.

```
public boolean search(List<T> a, T target) {  
    boolean found = false;  
    for (int i = 0; i < a.size(); i++) {  
        if (a.get(i).equals(target))  
            found = true;  
    }  
    return found;  
}
```

Version A

Which is more efficient?

```
public boolean search(List<T> a, T target) {  
    for (int i = 0; i < a.size(); i++) {  
        if (a.get(i).equals(target))  
            return true; ← “early exit”  
    }  
    return false;  
}
```

Version B

Efficiency

List search has a best, average, and worst case running time. If we timed both versions, what would you expect the data to look like for worst and average cases?

Version A (no early exit)

```
boolean search(List<T> a, T target){  
    boolean found = false;  
    for (int i = 0; i < a.size(); i++) {  
        if (a.get(i).equals(target))  
            found = true;  
    }  
    return found;  
}
```

No early exit

Worst case	4.75 ms
Average case	4.00 ms

Average time over four runs. List size = 100,000

Version B (early exit)

```
boolean search(List<T> a, T target) {  
    for (int i = 0; i < a.size(); i++) {  
        if (a.get(i).equals(target))  
            return true;  
    }  
    return false;  
}
```

Early exit

Worst case	4.70 ms
Average case	3.25 ms

Average time over four runs. List size = 100,000



??

Worst case	0.064 ms
Average case	0.065 ms

Average time over four runs. List size = 100,000

Improving efficiency

Tune/tweak the algorithm (e.g., early exit)

Apply a heuristic

Example: On each successful search, move the target to the first position in the list.

```
public boolean search(List<T> a, T target) {  
    for (int i = 0; i < a.size(); i++) {  
        T element = a.get(i);  
        if (element.equals(target)){  
            a.remove(element);  
            a.add(0, element);  
            return true;  
        }  
    }  
    return false;  
}
```

	No early exit	Early exit
Worst case	4.75 ms	4.70 ms
Average case	4.00 ms	3.25 ms

Assume there is a “working set” of search targets that follows the 90-10 rule. (90% of the searches are for only 10% of the elements.)

Average time of 10,000 runs with
 $N = 100,000$: **0.085ms**

Neither of these techniques improve worst case.

Improving worst case

Improving worst case efficiency often involves a more fundamental change to the algorithm or to the assumptions/constraints on the problem.

Current list search constraints: A list of elements that can be compared to each other.

20	12	35	15	10	80	30	17	2	1
0	1	2	3	4	5	6	7	8	9

New list search constraints: A list of elements that can be compared to each other, **arranged in non-decreasing order**.

1	2	10	12	15	17	20	30	35	80
0	1	2	3	4	5	6	7	8	9

This new constraint leads to a new algorithm that will be much more efficient than our current linear search.

Binary search

Solves the list search problem when the search space is ordered.

Assumes the input has been sorted into a monotonically increasing sequence: $[a_1 \leq a_2 \leq \dots a_n]$

Strategy:

Look at the middle element.

If it equals target, return.

If it is less than target, repeat on right half.

If it is greater than target, repeat on left half.

2	4	6	8	10	12	14
0	1	2	3	4	5	6

target = 15

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2	4	6	8	10	12	14
0	1	2	3	4	5	6

Binary search

Solves the list search problem when the search space is ordered.

Assumes the input has been sorted into a monotonically increasing sequence: $[a_1 \leq a_2 \leq \dots a_n]$

Strategy:

Look at the middle element.

If it equals target, return.

If it is less than target, repeat on right half.

If it is greater than target, repeat on left half.

2	4	6	8	10	12	14
0	1	2	3	4	5	6

target = 4

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2	4	6	8	10	12	14
0	1	2	3	4	5	6

Binary search

Solves the list search problem when the search space is ordered.

Assumes the input has been sorted into a monotonically increasing sequence: $[a_1 \leq a_2 \leq \dots a_n]$

Strategy:

Look at the middle element.

If it equals target, return.

If it is less than target, repeat on right half.

If it is greater than target, repeat on left half.

2	4	6	8	10	12	14
0	1	2	3	4	5	6

target = 5

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2	4	6	8	10	12	14
0	1	2	3	4	5	6

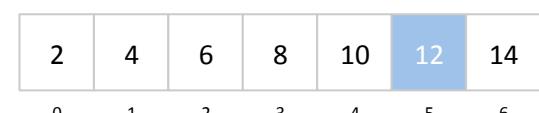
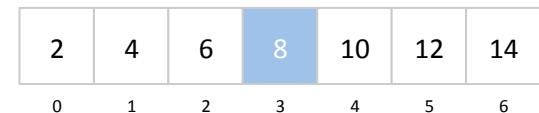
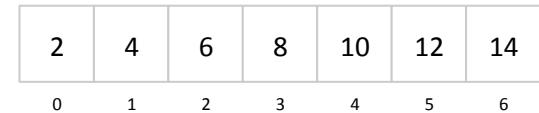
2	4	6	8	10	12	14
0	1	2	3	4	5	6

Binary search

```
public int search(int[] a, int target) {  
    int left = 0, right = a.length-1;  
    while (left <= right) {  
        int middle = (left + right) / 2;  
        if      (target < a[middle])           right = middle - 1;  
        else if (target > a[middle])           left = middle + 1;  
        else  
    }  
    return -1;  
}
```

Trace for target = 15

left	right	middle	a[middle]
0	6	3	8
4	6	5	12
6	6	6	14
7	6	/	/



Binary search

```
public int search(int[] a, int target) {  
    int left = 0, right = a.length-1;  
    while (left <= right) {  
        int middle = (left + right) / 2;  
        if      (target < a[middle])      right = middle - 1;  
        else if (target > a[middle])      left = middle + 1;  
        else                            return middle;  
    }  
    return -1;  
}
```

Trace for target = 4

left	right	middle	a[middle]
0	6	3	8
0	2	1	4

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2	4	6	8	10	12	14
0	1	2	3	4	5	6

Binary search

```
public int search(int[] a, int target) {  
    int left = 0, right = a.length-1;  
    while (left <= right) {  
        int middle = (left + right) / 2;  
        if      (target < a[middle])      right = middle - 1;  
        else if (target > a[middle])      left = middle + 1;  
        else                            return middle;  
    }  
    return -1;  
}
```

Trace for target = 5

left	right	middle	a[middle]
0	6	3	8
0	2	1	4
2	2	2	6
3	2	/	/

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2	4	6	8	10	12	14
0	1	2	3	4	5	6

2	4	6	8	10	12	14
0	1	2	3	4	5	6

Participation



Q: How many array elements will binary search examine when searching for the value 42 in the array below?

- A. 0
- B. 3
- C. 5
- D. 7
- E. 20

l:r:m
0:19:9
10:19:14
15:19:17
18:19:18
19:19:19

2	4	6	8	10	12	14	16	18	20	22	23	26	28	30	32	34	36	38	40
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Comparing efficiency

Linear Search

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

How many elements are eliminated by one comparison?

1

How many comparisons would be made in an array of size N?

N

How many *more* comparisons if the array doubled in size?

N

Comparing efficiency

Binary Search

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

2	4	6	8	10	12	14	16	18	20	22	24
0	1	2	3	4	5	6	7	8	9	10	11

How many elements are eliminated by one comparison?

half

How many comparisons would be made in an array of size N?

$\sim \log_2 N$

How many *more* comparisons if the array doubled in size?

1

Comparing efficiency

N	$\lg N$
16	4
32	5
64	6
128	7
256	8
512	9
1,024	10
2,048	11
4,096	12
8,192	13
16,384	14
32,768	15
65,536	16
131,072	17
262,144	18
524,288	19
1,048,576	20
2,097,152	21

