Sorting Algorithm

Examples of well-known algorithms

Nattee Niparnan

Goal

- Understand well-known algorithm
- Can write a working program of the algorithms
- Can analyze it

Problem

- Input:
  - Array A[1..n] of n data (we must be able to compare a pair of them)
- Output:
  - The same array but re-arranged such that A[i] <= A[i+1] for every i from 1 to N-1
- Example instance
  - Input: [1,5,3,2,7,1]
  - Output: [1,1,2,3,5,7]

Several Algorithms

Algorithm	Complexity
Bubble Sort (not-covered in this class)	$O(n^2)$
Selection Sort variation: Heap Sort	$\Theta(n^2)$ $O(n \lg n)$
Insertion Sort variation: Shellsort	$O(n^2)$
Radix Sort (already covered in Data Structure)	$O(n)$ (non-comparision)
Merge Sort (will be on D&C topics)	$O(n \lg n)$
Quick Sort (will be on D&C topics)	$O(n^2)$ (on average $O(n \log n)$)

Selection Sort

- Key Idea:
  - There are two parts of data: Unsorted array and Sorted array
    - Start by let the input be the unsorted array
    - let sorted array be an empty array
  - While the unsorted array is not empty
    - Get the maximum one
    - Put it at the front of the sorted array
    - Delete it from the unsorted array

Example pseudo code

```
#input: array A[1..n]
def selection_sort
  let B be an empty array
  while A is not empty
    #find the index of maximum item
    max_idx = 1
    for i = 1 to sizeof(A)
      if A[i] > A[max_idx]
        max_idx = i

    #move the maximum element
    insert A[max_idx] at the front of B
    remove item at position max_idx from A
  return B
```

What is the drawback of this pseudo code?

Can we convert it to a working C++?

How is the time complexity?

Example Code

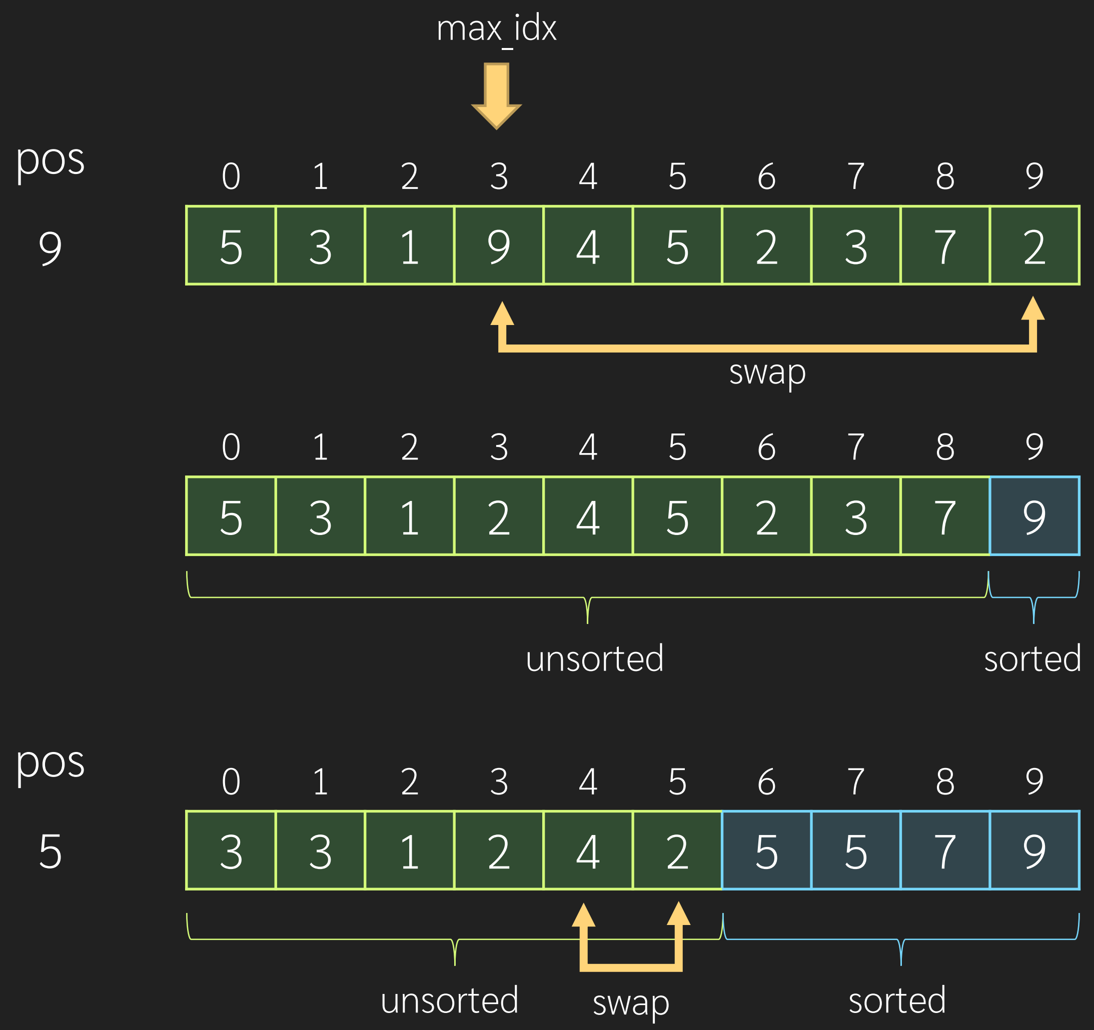
```
template<typename T>
vector<T> selection_sort(vector<T> A) {
  vector<T> B;
  while (A.size() > 0) {
    auto it = max_element(A.begin(),A.end());
    B.insert(B.begin(),*it);
    A.erase(it);
  }
  return B;
}
```

```
template<typename T>
vector<T> selection_sort(vector<T> A) {
  vector<T> B(A.size());
  while (A.size() > 0) {
    auto it = max_element(A.begin(),A.end());
    B[A.size() - 1] = *it;
    swap(*it, A[a.size()-1]);
    A.pop_back();
  }
  return B;
}
```

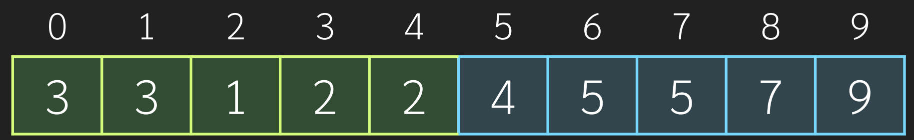
Which one is faster?

Actual Code

```
template<typename T>
void selection_sort(vector<T> &A) {
  size_t pos = A.size()-1;
  for ( ; pos > 0 ; pos--) {
    int max_idx = 0;
    for (size_t i = 0;i <= pos;i++) {
      if (A[i] > A[max_idx]) {
        max_idx = i;
      }
    }
    swap(A[pos],A[max_idx]);
  }
}
```



- In-place swap
  - Does not need another vector
  - O(1) in swap



Complexity Analysis

```
#input: array A[1..n]
def selection_sort
  pos = n
  while pos > 1
    #find the index of maximum item
    max_idx = 1
    for i = 1 to pos
      if A[i] > A[max_idx]
        max_idx = i

    #swap the maximum element
    swap(A[max_idx],A[pos])
    pos = pos - 1
  return A
```

- While-loop runs n-1 times
- In each loop, the for-loop runs pos times
- T(n) = n + n-1 + n-2 + … + 2
- T(n) = $\Theta(n^2)$

Selection Sort variation: Heap Sort

- Improve finding max element by using Binary Heap
- Key Idea:
  - Treat unsorted portion of the array as a binary heap
    - We can find max and remove very fast
  - At start, we build_heap ($O(n)$) on the unsorted array
    - Build heap = fix_down on n/2 to 1
  - For each iteration, assume that the heap is at A[1..pos]
    - the maximum element in [1..pos] is at A[1]
    - Do Binary Heap's pop, which is swapping A[1] with A[pos] and fix_down (same thing as we want)

Heap Sort

```
#input: array A[1..n]
def heap_sort(A)
  pos = n
  for i = n/2 to 1
    #fix_down will perform heap's fix_down at position i
    #  assuming the heap is in the array A[1..pos]
    fix_down(A,i,pos);

  while pos > 1
    swap(A[1],A[n])
    pos = pos - 1
    fix_down(A,1,pos);
  return A
```

```
#input: A = array A
#input: i is the position to fix down
#input: size is the size of heap in A (A[1..size])
def fix_down(A,i,size)
  tmp = A[i];
  while ((c = 2 * idx + 1) < mSize)
    if (c + 1 <= size && A[c] < A[c+1]) c++;
    if (A[c] < tmp ) break;
    A[i] = A[c];
    i = c;
  A[i] = tmp
```

Heap Sort Analysis

- build_heap requires O(N)
  - As we have learned from priority_queue in our Data Structure class
- There is n-1 iterations of the while loop
  - Each loop is basically pop() operation of the binary heap which is O(lg n)
- This total to O(n log n)

Insertion Sort

- Key Idea:
  - Maintain two-parts of the input array, unsorted and sorted part, similar to the actual version of the selection sort
  - In each iteration, instead of identifying the maximum element in the unsorted part and prepend it to the sorted part
    - Insertion sort try to insert the last element of the unsorted part to the sorted part so that the new element is at the correct position (making the sorted list still sorted)

Pseudo-code

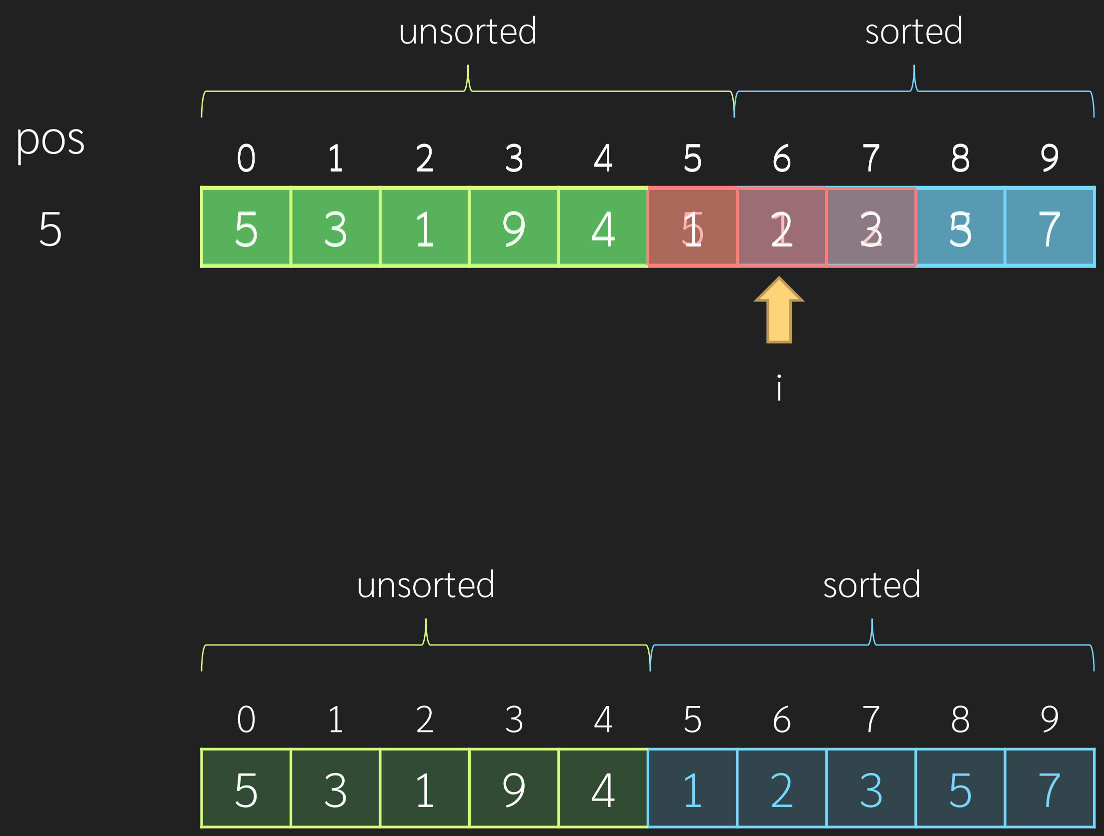
```
#input: array A[1..n]
def insertion_sort
  pos = n-1
  while pos > 0
    #find minimum i in the range [pos+1..n] such that
    #  A[pos] <= A[i]
    #  if no such i exists, let i be n
    i = pos+1
    while (i <= n && A[pos] > A[i])
      i = i + 1

    #insert A[pos] after A[i]
    j = pos
    tmp = A[pos]
    while (j < i)
      A[j] = A[j+1];
      j = j + 1
    A[i] = pos

    pos = pos - 1
```

Actual Code

```
template <typename T>
void insertion_sort(vector<T> &A) {
  for (int pos = A.size()-2;pos >= 0;pos--) {
    T tmp = A[pos];
    size_t i = pos+1;
    while (i < A.size() && A[i] < tmp) {
      A[i-1] = A[i];
      i++;
    }
    A[i-1] = tmp;
  }
}
```



- Find and insert simultaneously

Analysis

```
template <typename T>
void insertion_sort(vector<T> &A) {
  for (int pos = A.size()-2;pos >= 0;pos--) {
    T tmp = A[pos];
    size_t i = pos+1;
    while (i < A.size() && A[i] < tmp) {
      A[i-1] = A[i];
      i++;
    }
    A[i-1] = tmp;
  }
}
```

- For loop N-1 iteration
- Each for loop perform at most (pos-1)
- T(N) = 1+2+3+…N-1
- It is possible that T(N) = 1+1+1+…+1
  - When?

Insertion Sort is $O(N^2)$

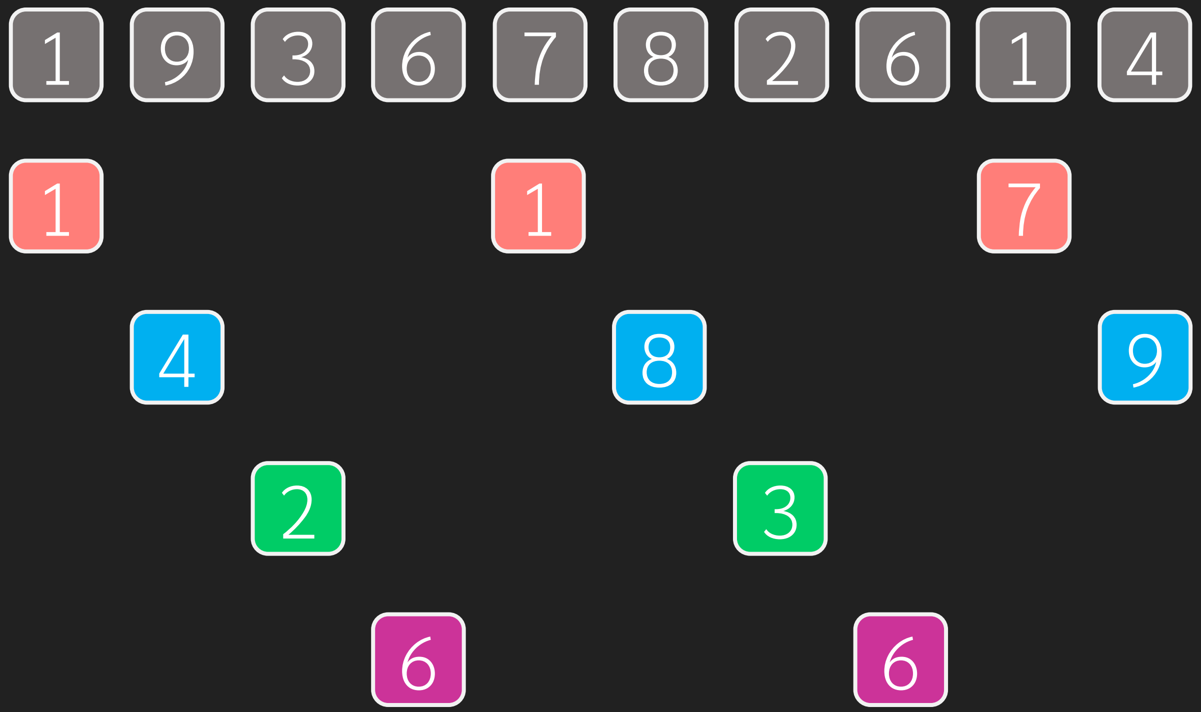
Analysis of insertion sort

- On each iterations, insertion sorts try to insert A[pos] to its correct position
  - If the correct position of A[pos] is far from pos, we need to move it that much
- If we can quickly identify the correct position of A[pos], we still need to move that many elements because of insert
- When the input is almost sorted, i.e., their elements does not stay away from its correct position, insertion sort works fast
  - For example [1, 2, 3, 2, 4, 7, 5, 6]

Insertion Sort variation: Shellsort

- Invented by Donald shell
- Key Idea:
  - Start by letting G = some large value less than N and doing these process
    - (a) Divide A into G smaller arrays, each consist of element in A that is G elements apart
    - (b) Sort each sub-array by insertion sort
  - Repeat (a) and (b) with smaller value of G
    - Keep doing (a) and (b) until G is 1
    - When G = 1 it is basically the insertion sort but the array should be almost sorted
- It is like trying to move each element to its correct position faster
- Sequence of G is important

Example when G = 4

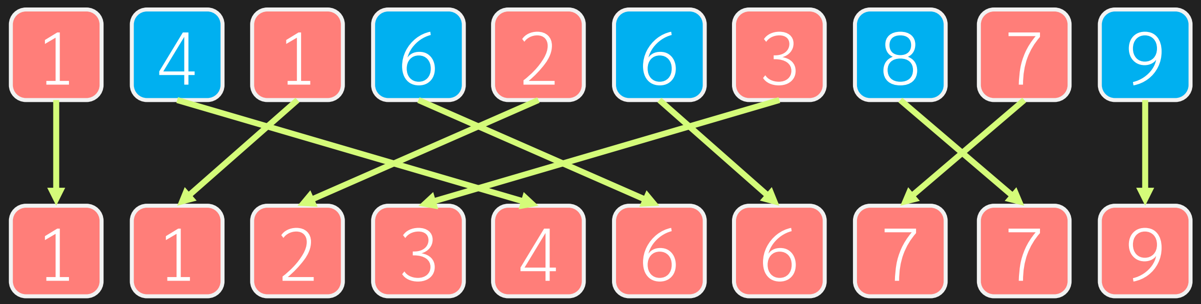


(a) Divide into 4 groups

(b) Sort each group

Do it again with G = 2

Do it again with G = 1



Pseudo-code

```
#input: array A[1..n]
def shell_sort
  gaps = [701,301,132,57,23,10,4,1]
  for G in gaps

    #for each value of G, we do G groups
    for round = n downto n-gap+1
      #in each group, we just do the insertion sort
      #  where each element is G item aparts
      pos = round-G
      while pos > 0
        tmp = A[pos]
        i = pos+G
        while (i <= n && tmp > A[i])
          A[i-G] = A[i]
          i = i + G
        A[i-G] = tmp

        pos = pos - 1
```

- Code can be shorter

Analysis

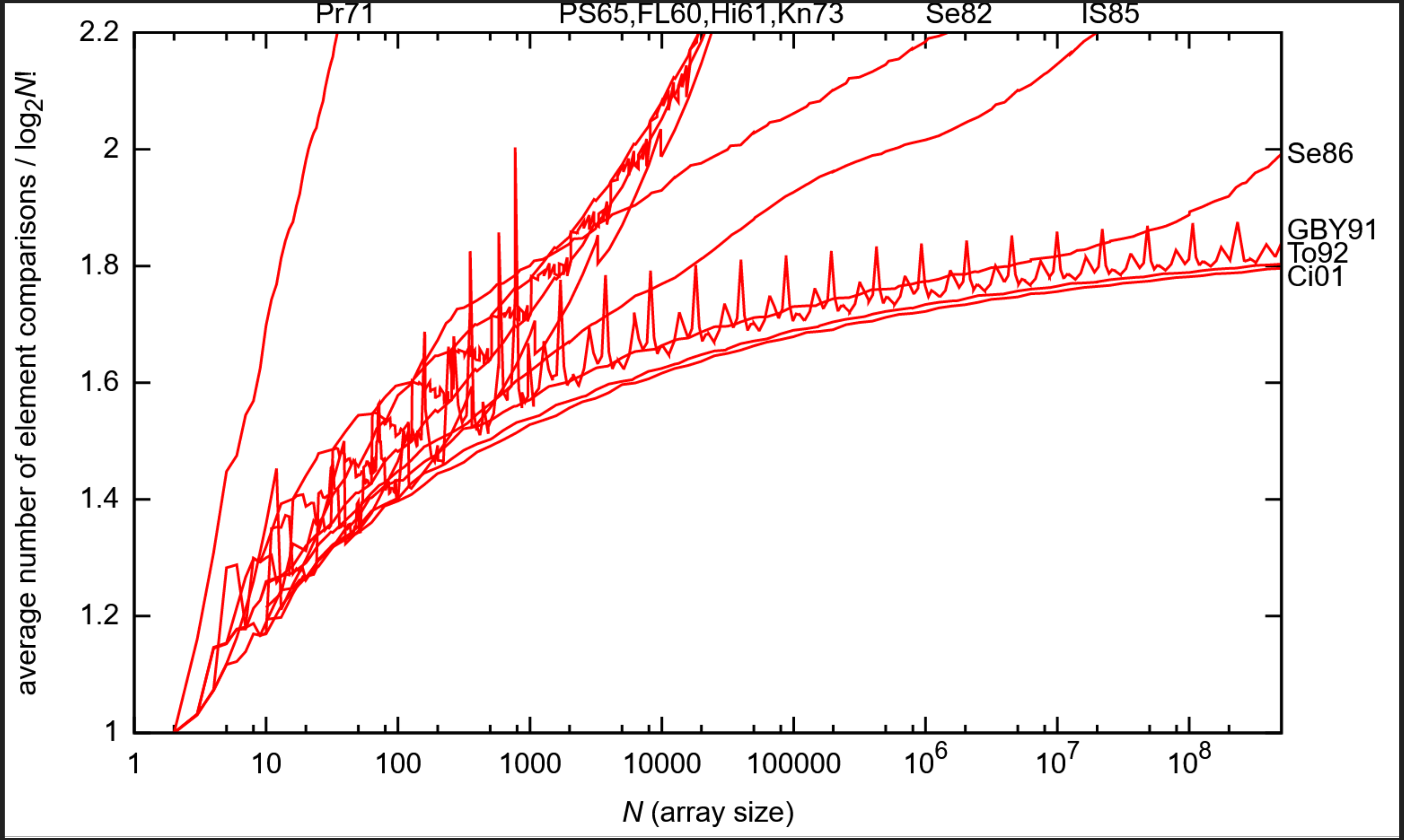
- Very hard
- We know that when G = 1, it is basically the insertion sort
  - So it is $O(N^2)$ with $\Theta(N)$ best case
  - G = 1 is required at the last step so we ca guarantee that the array is sorted
- The performance actually depends on the sequence of G

Analysis

OEIS	General term ($k \ge 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
	$\left\lfloor \frac{N}{2^k} \right\rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta\left(N^2\right)$ [e.g. when $N = 2^p$]	Shell, 1959[4]
	$2\left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta\left(N^{\frac{3}{2}}\right)$	Frank & Lazarus, 1960[8]
A000225	$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Hibbard, 1963[9]
A083318	$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Papernov & Stasevich, 1965[10]
A003586	Successive numbers of the form $2^p 3^q$ (3-smooth numbers)	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta\left(N \log^2 N\right)$	Pratt, 1971[1]
A003462	$\frac{3^k - 1}{2}$, not greater than $\left\lceil \frac{N}{3} \right\rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Knuth, 1973,[3] based on Pratt, 1971[1]
A036569	$\prod_I a_q$, where $a_0 = 3$, $a_q = \min\left\{n \in \mathbb{N} : n \ge \left(\frac{5}{2}\right)^{q+1}, \forall p : 0 \le p < q \Rightarrow \gcd(a_p, n) = 1\right\}$, $I = \left\{0 \le q < r \mid q \ne \frac{1}{2}\left(r^2 + r\right) - k\right\}$, $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$	$1, 3, 7, 21, 48, 112, \ldots$	$O\left(N^{1+\sqrt{\frac{8\ln(5/2)}{\ln(N)}}}\right)$	Incerpi & Sedgewick, 1985,[11] Knuth[3]
A036562	$4^k + 3 \cdot 2^{k-1} + 1$, prefixed with 1	$1, 8, 23, 77, 281, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1982[6]
A033622	$\begin{cases} 9\left(2^k - 2^{\frac{k}{2}}\right) + 1 & k \text{ even}, \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1 & k \text{ odd} \end{cases}$	$1, 5, 19, 41, 109, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1986[12]
	$h_k = \max\left\{\left\lfloor \frac{5h_{k-1}}{11} \right\rfloor, 1\right\}, h_0 = N$	$\left\lfloor \frac{5N}{11} \right\rfloor, \left\lfloor \frac{5}{11}\left\lfloor \frac{5N}{11} \right\rfloor \right\rfloor, \ldots, 1$	Unknown	Gonnet & Baeza-Yates, 1991[13]
A108870	$\left\lceil \frac{1}{5}\left(9 \cdot \left(\frac{9}{4}\right)^{k-1} - 4\right) \right\rceil$	$1, 4, 9, 20, 46, 103, \ldots$	Unknown	Tokuda, 1992[14]
A102549	Unknown (experimentally derived)	$1, 4, 10, 23, 57, 132, 301, 701$	Unknown	Ciura, 2001[15]

- From Wikipedia
- Current best case $O(N^{4/3})$
- Ciura's sequence perform best, empirically

Analysis



- Also from wikipedia