Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Philip Bille

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Algorithms and Data Structures

- Algorithmic problem. Precisely defined relation between input and output.
- Algorithm. Method to solve an algorithmic problem.
  - Discrete and unambiguous steps.
  - Mathematical abstraction of a program.
- Data structure. Method for organizing data to enable queries and updates.

Example: Find Max

- Find max. Given an array A[0..n-1], find an index i, such that A[i] is maximal.
  - Input. Array A[0..n-1].
  - Output. An index i such that A[i] ≥ A[j] for all indices j ≠ i.
- Algorithm.
  - Process A from left to right and maintain value and index of maximal value seen so far.
  - Return index.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

Description of Algorithms

- Natural language.
  - Process A from left-to-right and maintain value and index of maximal value seen so far.
  - Return index.
- Program.
- Pseudocode.

```
def findMax(A):
    m = 0
    for i in range(len(A)):
        if A[i] > A[m]:
            m = i
    return m
```

```
FINDMAX(A, n)
   max = 0
   for i = 0 to n-1
      if (A[i] > A[max]) max = i
   return max
```

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Peaks

- Peak. A[i] is a peak if A[i] is as least as large as it's neighbors:
- A[i] is a peak if A[i-1] ≤ A[i] ≥ A[i+1] for i ∊ {1, .., n-2}
- A[0] is a peak if A[0] ≥ A[1].
- A[n-1] is a peak if A[n-2] ≤ A[n-1].

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

- Peak finding. Given an array A[0..n-1], find an index i such that A[i] is a peak.
  - Input. A array A[0..n-1].
  - Output. An index i such that A[i] is a peak.

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Algorithm 1

- Algorithm 1. For each entry check if it is a peak. Return the index of the first peak.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

- Pseudocode.

```
PEAK1(A, n)
   if A[0] ≥ A[1] return 0
   for i = 1 to n-2
      if A[i-1] ≤ A[i] ≥ A[i+1] return i
   if A[n-2] ≤ A[n-1] return n-1
```

- Challenge. How do we analyze the algorithm?

Theoretical Analysis

- Running time/time complexity.
  - T(n) = number of steps that the algorithm performs on input of size n.
- Steps.
  - Read/write to memory (x := y, A[i], i = i + 1, ...)
  - Arithmetic/boolean operations (+, -, *, /, %, &&, ||, &, |, ^, ~)
  - Comparisons (<, >, =<, =>, =, ≠)
  - Program flow (if-then-else, while, for, goto, function call, ..)
- Worst-case time complexity. Maximal running time over all inputs of size n.

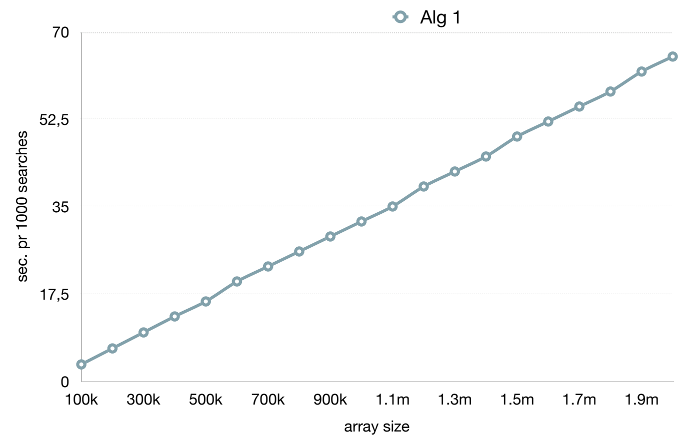
Theoretical Analysis

- Running time. What is the running time T(n) for algorithm 1?

```
PEAK1(A, n)
   if A[0] ≥ A[1] return 0              c1
   for i = 1 to n-2                     (n-2)·c2
      if A[i-1] ≤ A[i] ≥ A[i+1] return i
   if A[n-2] ≤ A[n-1] return n-1        c3
```

$T(n) = c_1 + (n-2) \cdot c_2 + c_3$

- T(n) is a linear function of n: T(n) = an + b
- Asymptotic notation. T(n) = O(n)
- Experimental analysis.
  - What is the experimental running time of algorithm 1?
  - How does the experimental analysis compare to the theoretical analysis?



Peaks

- Algorithm 1 finds a peak in O(n) time.
- Theoretical and experimental analysis agrees.
- Challenge. Can we do better?

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Algorithm 2

- Observation. A maximal entry A[i] is a peak.
- Algorithm 2. Find a maximal entry in A with FINDMAX(A, n).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

```
FINDMAX(A, n)
   max = 0
   for i = 0 to n-1
      if (A[i] > A[max]) max = i
   return max
```

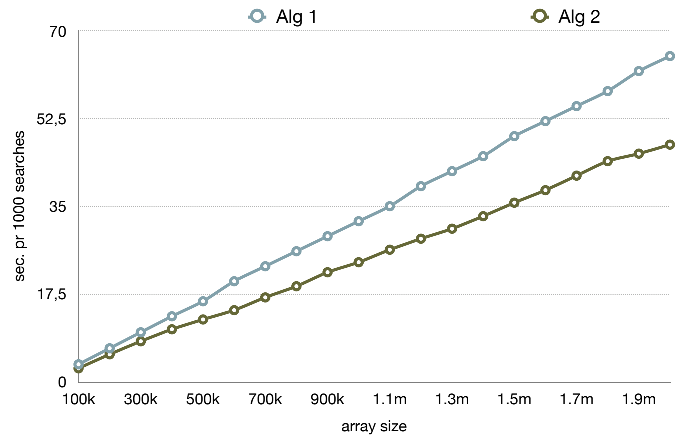
Theoretical Analysis

- Running time. What is the running time T(n) for algorithm 2?

```
FINDMAX(A, n)
   max = 0                              c4
   for i = 0 to n-1                     n·c5
      if (A[i] > A[max]) max = i
   return max                           c6
```

$T(n) = c_4 + n \cdot c_5 + c_6 = O(n)$

- Experimental analysis. Better constants?



Peaks

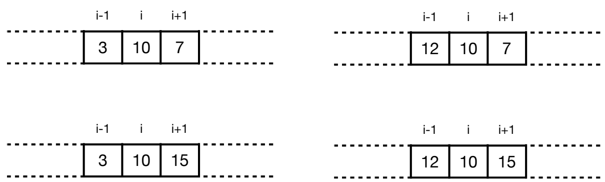
- Theoretical analysis.
  - Algorithm 1 and 2 find a peak in O(n) time.
- Experimental analysis.
  - Algorithm 1 and 2 run in O(n) time in practice.
  - Algorithm 2 is a constant factor faster than algorithm 1.
- Challenge. Can we do significantly better?

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3

Algorithm 3

- Clever idea.
  - Consider any entry A[i] and it's neighbors A[i-1] and A[i+1].
  - Where can a peak be relative to A[i]?
    - Neighbor are ≤ A[i] ⟹ A[i] is a peak.
    - Otherwise A is increasing in at least one direction ⟹ peak must exist in that direction.



- Challenge. How can we turn this into a fast algorithm?

Algorithm 3

- Algorithm 3.
  - Consider the middle entry A[m] and neighbors A[m-1] and A[m+1].
  - If A[m] is a peak, return m.
  - Otherwise, continue search recursively in half with the increasing neighbor.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

Algorithm 3

- Algorithm 3.
  - Consider the middle entry A[m] and neighbors A[m-1] and A[m+1].
  - If A[m] is a peak, return m.
  - Otherwise, continue search recursively in half with the increasing neighbor.

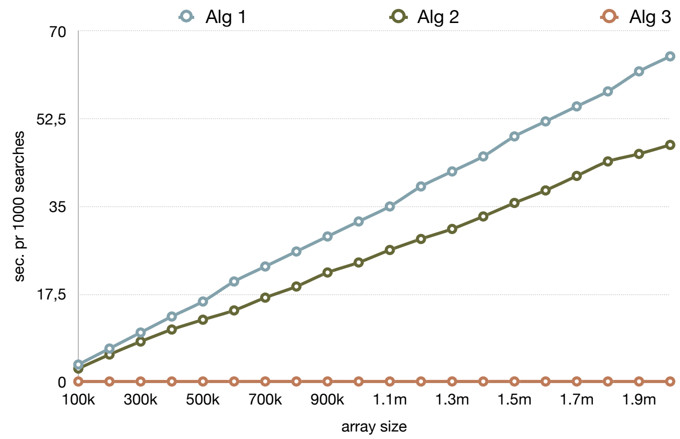
```
PEAK3(A,i,j)
   m = ⌊(i+j)/2)⌋
   if A[m] ≥ neighbors return m
   elseif A[m-1] > A[m]
      return PEAK3(A,i,m-1)
   elseif A[m] < A[m+1]
      return PEAK3(A,m+1,j)
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	7	15	17	11	2	3	6	8	7	5	9	5	23

Algorithm 3

```
PEAK3(A,i,j)
   m = ⌊(i+j)/2)⌋
   if A[m] ≥ neighbors return m
   elseif A[m-1] > A[m]
      return PEAK3(A,i,m-1)
   elseif A[m] < A[m+1]
      return PEAK3(A,m+1,j)
```

- Running time.
- A recursive call takes constant time.
- How many recursive calls?
- A recursive call halves the size of the interval. We stop when the array has size 1.
  - 1st recursive call: $n/2$
  - 2nd recursive call: $n/4$
  - ....
  - kth recursive call: $n/2^k$
  - ....
- ⟹ After ~$\log_2 n$ recursive call array has size ≤ 1.
- ⟹ Running time is $O(\log n)$
- Experimental analysis. Significantly better?



Peaks

- Theoretical analysis.
  - Algorithm 1 and 2 finds a peak in O(n) time.
  - Algorithm 3 finds a peak in O(log n) time.
- Experimental analysis.
  - Algorithm 1 and 2 run in O(n) time in practice.
  - Algorithm 2 is a constant factor faster than algorithm 1.
  - Algorithm 3 is much, much faster than algorithm 1 and 3.

Introduction

- Algorithms and Data Structures
- Peaks
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3