Algorithms: Analysis & Design

40354

What are we going to do?

- What are algorithms?
- What is the course about?
- A short look to the term.
- Grading Policy!
- Design and analysis of algorithms.
- Solving some warm-up problems!

Algorithms

Algorithm.

- [webster.com] A procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation.

- [Knuth, TAOCP] An algorithm is a finite, definite, effective procedure, with some input and some output.

Great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing. - *Francis Sullivan*

Etymology

Etymology. [Knuth, TAOCP]

- *Algorism* = process of doing arithmetic using Arabic numerals.
- A misperception: *algiros* [painful] + *arithmos* [number].
- True origin: Abu 'Abd Allah Muhammad ibn Musa al-Khwarizm was a famous 9th century Persian textbook author who wrote *Kitab al-jabr wa'l-muqabala*, which evolved into today's high school algebra text.

Why we analyze algorithms?

- Performance Prediction

  The first goal is to explain or predict the empirical performance of algorithms. Frequently this goal is pursued for a fixed algorithm like quick sort.
- Identify Optimal Algorithms

  The second goal is to rank different algorithms according to their performance, and ideally to single out one algorithm as "optimal."
- Design New Algorithms

Algorithmic Paradigms

Design and analysis of computer algorithms.

- Desing by induction.
- Divide-and-conquer.
- Greedy.
- Dynamic programming.
- Network flow.
- Linear Programming
- Intractability.
- Coping with intractability.

Critical thinking and problem-solving.

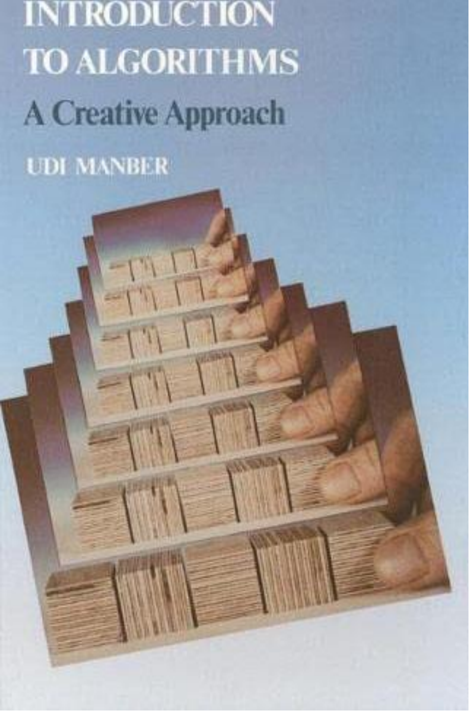
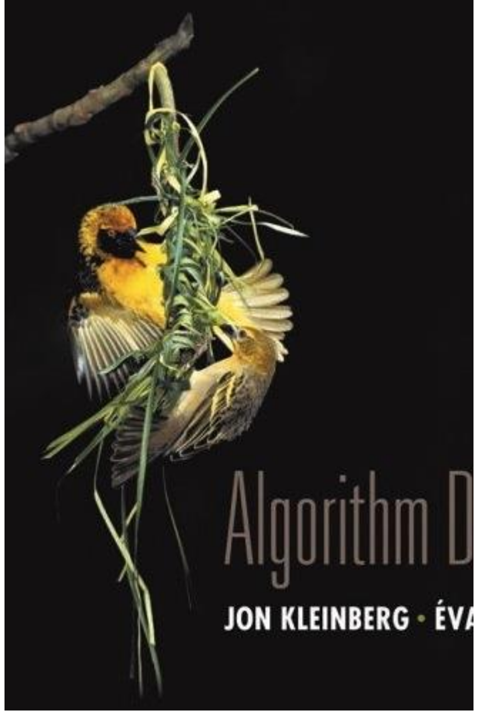
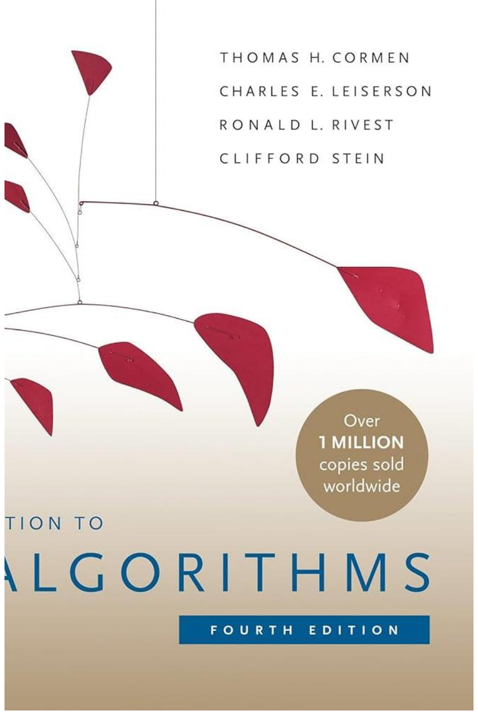
Logistics Grading Policy

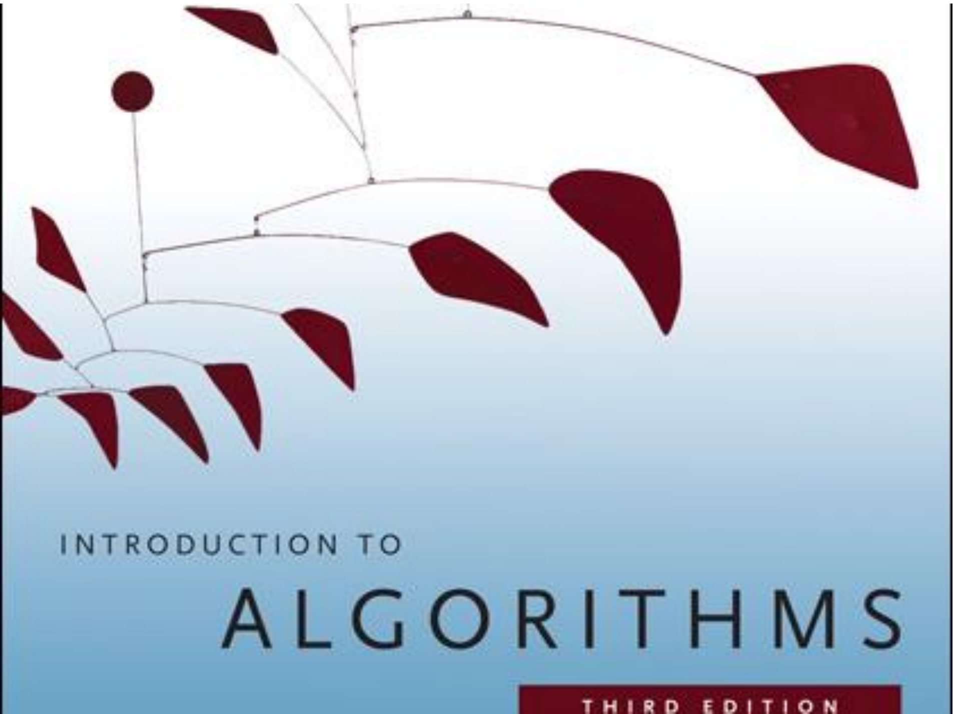
Grading:

- Mid-Term 1 (6 pts) – 30st Farvardin 9 am. (End of Greedy)
- Mid-Term 2 (6 pts) – 27th Ordibehesht 9 am. (Graph + Flow)
- Final Exam (6 pts) 27th Khordad 15:30 am. (Rest..)
- Programming Project (3 pts)
- ICPC (+1 pts)

To this end:

- 4 Assignment (without delivery)
- 2 TA classes (1 on-site + recorded videos)
- Similar questions from homework in exams. (about 25% of each exam is very similar to assginments.


INTRODUCTION
TO ALGORITHMS
A Creative Approach
UDI MANBER

Algorithm D
JON KLEINBERG · ÉVA

THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
Over 1 MILLION copies sold worldwide
TION TO
LGORITHMS
FOURTH EDITION

Algorithm
FOURTH EDIT
ROBERT SEDGEWICK | KEVIN WA

INTRODUCTION TO
ALGORITHMS
THIRD EDITION

References

Applications

Wide range of applications.

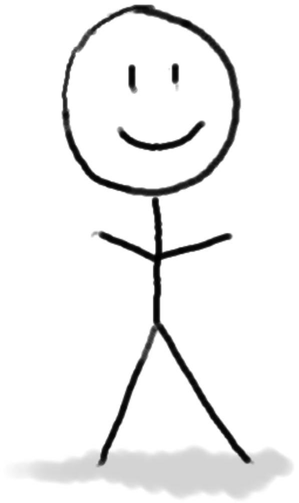
- Caching.
- Compilers.
- Databases.
- Scheduling.
- Networking.
- Data analysis.
- Signal processing.
- Computer graphics.
- Scientific computing.
- Operations research.
- Artificial intelligence.
- Computational biology.
- . . .

Our guiding questions:

Does it work?

Is it fast?

Can I do better?



Our internal monologue…

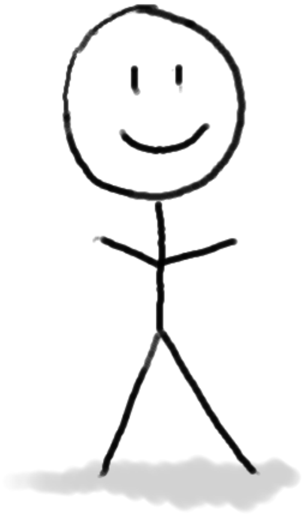
What exactly do we mean by better? And what about that corner case? Shouldn't we be zero-indexing?

Does it work?

Is it fast?

Can I do better?

Dude, this is just like that other time. If you do the thing and the stuff like you did then, it'll totally work real fast!



Plucky the
Pedantic Penguin

Detail-oriented
Precise
Rigorous

Lucky the
Lackadaisical Lemur

Big-picture
Intuitive
Hand-wavey

Both sides are necessary!

Algorithm Analysis

Analysis refers to mathematical techniques for establishing both the correctness and efficiency of algorithms.

Efficiency: Given an algorithm A, we want to know how efficient it is. This includes several possible criteria:

- What is the asymptotic complexity of algorithm A?
- How does the average-case complexity of A compare to the worst-case complexity?
- Is A the most efficient algorithm to solve the given problem? (For example, can we find a lower bound on the complexity of any algorithm to solve the given problem?)

Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

Def. An algorithm is poly-time if the below property holds.

> There exists constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by $c N^d$ steps.

Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size N.

- Generally captures efficiency in practice.
- Not data model is assumed.
- Mathematical tractability.
- Good upper bounds are awesome.
- BUT, it is very pessimistic and hence can rank algorithms inaccurately.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.

Worst-Case Polynomial-Time

Def. An algorithm is practical if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.

Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

Why It Matters

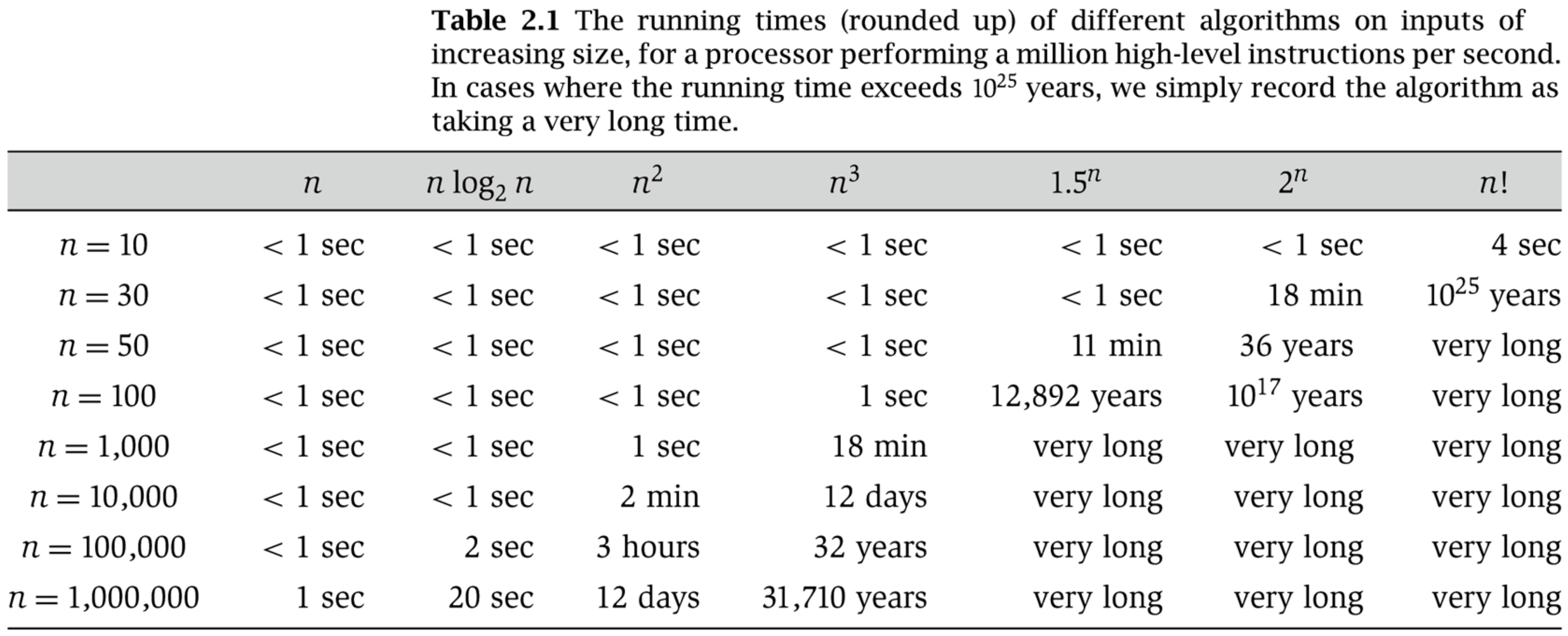


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Fibonacci Series

- General Form

$T(n) = T(n-1) + T(n-2)$

$T(0) = 0$

$T(1) = 1$

Three Different Algorithms:

- Recursive Algorithm
- Iterative Algorithm
- One More Interesting!

Asymptotic Order of Growth

Asymptotic Order of Growth

Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Notation

Slight abuse of notation. $T(n) = O(f(n))$.

- Not transitive:
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - but $f(n) \neq g(n)$.
- Better notation: $T(n) \in O(f(n))$.

Meaningless statement. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.

- Use Ω for lower bounds.

Properties

Transitivity.

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivity.

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
- If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$.

Asymptotic Bounds for Some Common Functions

Polynomials. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$.

$\uparrow$

can avoid specifying the base

Logarithms. For every $x > 0$, $\log n = O(n^x)$.

$\uparrow$

log grows slower than every polynomial

Exponentials. For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.

$\uparrow$

every exponential grows faster than every polynomial

A Survey of Common Running Times

Linear Time: O(n)

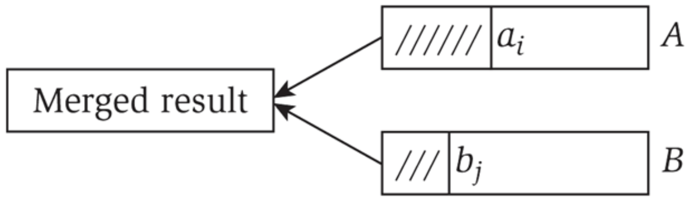
Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a_1
for i = 2 to n {
   if (a_i > max)
      max ← a_i
}
```

Linear Time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.



```
i = 1, j = 1
while (both lists are nonempty) {
   if ($a_i \le b_j$) append $a_i$ to output list and increment i
   else               append $b_j$ to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes O(n) time.
Pf. After each comparison, the length of output list increases by 1.

O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms.

↑ also referred to as linearithmic time

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane (x_1, y_1), …, (x_n, y_n), find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← $(x_1 - x_2)^2 + (y_1 - y_2)^2$
for i = 1 to n {
    for j = i+1 to n {
        d ← $(x_i - x_j)^2 + (y_i - y_j)^2$
        if (d < min)
            min ← d
    }
}
```

← don't need to take square roots

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion.

Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

```
foreach set S_i {
   foreach other set S_j {
      foreach element p of S_i {
         determine whether p also belongs to S_j
      }
      if (no element of S_i belongs to S_j)
         report that S_i and S_j are disjoint
   }
}
```

Polynomial Time: $O(n^k)$ Time

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

↖ k is a constant

$O(n^k)$ solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set = $O(k^2)$.
- Number of k element subsets = $\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \leq \frac{n^k}{k!}$
- $O(k^2\, n^k / k!) = O(n^k)$.

↖ poly-time for k=17, but not practical

Exponential Time

Independent set. Given a graph, what is maximum size of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* ← φ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

References

References

- Sections 2.1, 2.2, and 2.4 of the text book "algorithm design" by Jon Kleinberg and Eva Tardos
- The original slides were prepared by Kevin Wayne. The slides are distributed by Pearson Addison-Wesley.

Algorithm Design

Design refers to general strategies for creating new algorithms. If we have good design strategies, then it will be easier to end up with correct and efficient algorithms.

Here are some examples of useful design strategies

- reductions
- divide-and conquer
- greedy
- dynamic programming
- local search
- exhaustive search (backtracking, branch-and-bound)

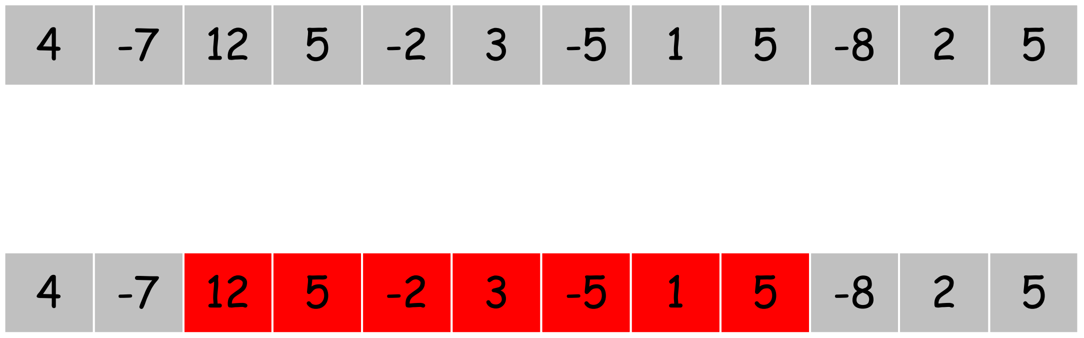
Maximum Sum Subarray

Maximum Sum Subarray

Problem: Given a one dimensional array A[1..n] of numbers. Find a contiguous subarray with largest sum within A.

Assume an empty subarray has sum 0.

Example:



Algorithm 1 (brute-force)

```
sol = 0
for i = 1 to n do
    for j = i to n do
        sum = 0
        for k = i to j do
            sum = sum + a[k]
        if sum > sol then
           sol = sum
return sol
```

Running time: $T(n) = O(n^3)$

Algorithm 2 (brute-force)

Observation: Let S[i] = A[1]+…+A[i]. We have A[i]+…+A[j]=S[j]-S[i-1]

```
Pre-Processing
S[0] = 0
for i = 1 to n do
    S[i] = S[i-1]+A[i]
```

Running time of pre-processing: $T(n) = O(n)$

```
sol = 0
for i = 1 to n do
    for j = i to n do
        if S[j]-S[i-1] > sol then
            sol = S[j]-S[i-1]
return sol
```

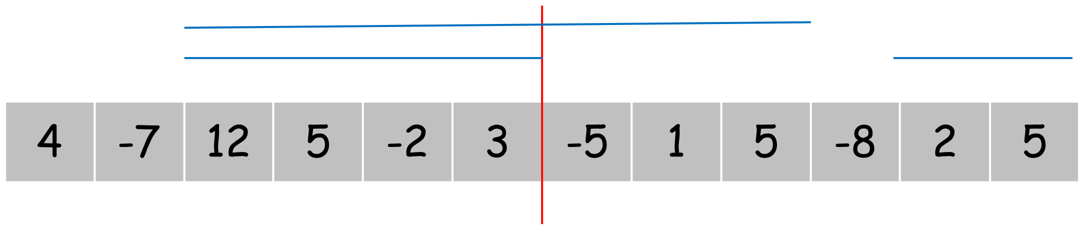
Running time: $T(n) = O(n^2)$

Algorithm 3 (divide and conquer)

The general strategy: Divide into 2 equal-size subarrays

Case 1: optimal solution is in one subarray

Case 2: optimal solution crosses the splitting line

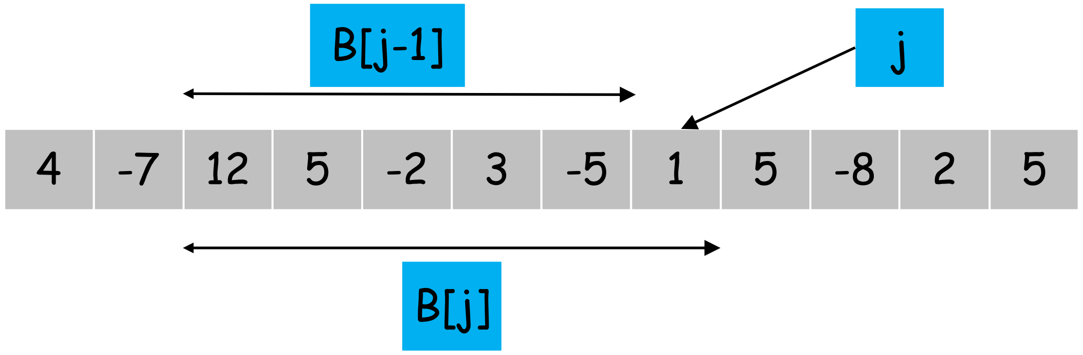


```
MCS(A[1..n])
if n = 1 then return max(0, a[1])
sol = max(MCS(A[1…n/2]), MCS(A[n/2+1…n])
Lsol = 0
for i = n/2 downto 1 do
    if S[n/2]-S[i-1] > Lsol then
       Lsol = S[n/2]-S[i-1]
Rsol = 0
for i = n/2+1 to n do
    if S[i]-S[n/2-1] > Rsol then
       Rsol = S[i]-S[n/2-1]
return max(sol, Lsol+Rsol)
```

Running time: $T(n) = 2T\left(\frac{n}{2}\right) + O(n) \rightarrow T(n) = O(n \log n)$

Algorithm 4 (dynamic programming)

The general strategy: consider the subarray with max sum ending at index j. Let B[j] be the sum of all entries in this subarray. The output is max(B[j]) over all j=1,…,n. B[j] can be computed from B[j-1] by the recursive formula B[j]=max(0,A[j], A[j]+B[j-1])



```
sol = 0
B[0] = 0
for i = 1 to n do
    B[i] = max(0,A[i], A[i]+B[i-1])
    if B[i] > sol then
      sol = B[i]
return sol
```

Running time: $T(n) = O(n)$

3-Sum

3-Sum

Problem: Given a array $A[1..n]$ of numbers.

Do there exist three distinct numbers in A whose sum equals 0?

Algorithm 1 (brute-force)

```
for i = 1 to n-2 do
    for j = i+1 to n-1 do
        for k = j+1 to n-2 do
           if a[i]+a[j]+a[k] = 0 then
              return yes
return no
```

- Running time: $T(n) = O(n^3)$

Algorithm 2

Observation: instead of having three nested loops, we have two nested loops with index i and j and then we search for an A[k] for which A[i]+A[j]+A[k] =0

```
Sort A
for i = 1 to n-2 do
    for j = i+1 to n-1 do
        search -A[i]-A[j] in A[j+1,…,n]
        if search is successful then
             return yes
return no
```

Running time: $O(n^2 \log n)$

Algorithm 3

Observation: Simultaneously scan from both ends of A looking for A[j] + A[k] = −A[i]. At any step of the algorithm, we either increase j or decrease k.

```
Sort A
for i = 1 to n-2 do
    j = i+1
    k = n
    while j < k do
       if A[i]+A[j]+A[k] < 0 then j = j+1
       else A[i]+A[j]+A[k] > 0 then k = k-1
       else return yes
return no
```

Running time: $O(n^2)$

- We have only two nested loops and each step of these loops takes O(1) time

Still it is not clear whether there is a better solution.

References

References

- Section 5.8 of the text book “introduction to algorithms: a creative approach” by Udi Manber, 1989.
- Section 4.1 of the text book “introduction to algorithms” by CLRS, 3rd edition.
- 3-Sum section is from slides prepared by Douglas R. Stinson.