Algorithm Design

- Maximum Sum Subarray
- 3-SUM

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:

4	-7	12	5	-2	3	-5	1	5	-8	2	5

4	-7	12	5	-2	3	-5	1	5	-8	2	5

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
```

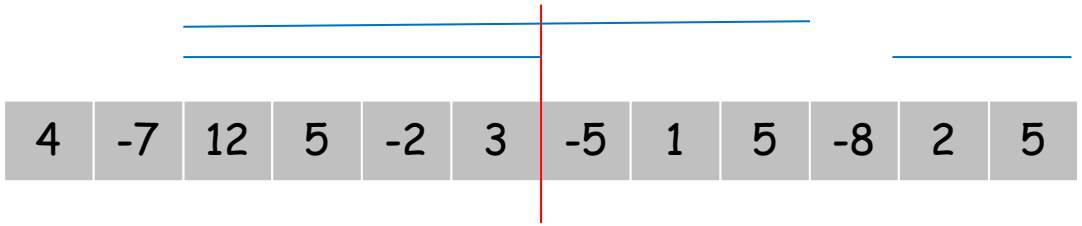
Running time: $T(n) = O(n^2)$

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

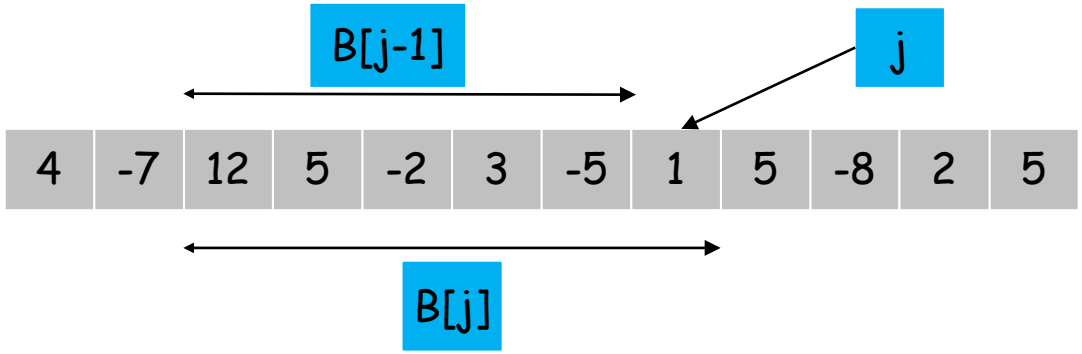



```
MCS(A[1..n])
if n = 1 then return max(0, a[1])
sol = max(MCS(A[1…n/2]), MCS(A[n/2+1…n])
Lsol = 0
for i = n/2 downto 1 do
    if S[n/2]-S[i-1] > Lsol then
       Lsol = S[n/2]-S[i-1]
Rsol = 0
for i = n/2+1 to n do
    if S[i]-S[n/2-1] > Rsol then
       Rsol = S[i]-S[n/2-1]
return max(sol, Lsol+Rsol)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

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.