roadmap

'in between math and programming’
understanding of classical algorithms and data structures
understanding of classical programming paradigms
divide and conquer, dynamic programming, greedy algorithms
be able to analyze time complexity and space complexity
smart choices for data may improve the performance of an algorithm
be able to design new algorithms and reason about them

overview

• make change
• max-subarray
• rod cutting
• knapsack 0/1
• material
making change: daily life

make 4, 35 euro with a minimum number of coins:

as many 2 as possible: $2 + 2$

as many 0, 20 as possible: $2 + 2 + 0, 2$

as many 0, 1 as possible: $2 + 2 + 0, 2 + 0, 1$

as many 0, 05 as possible: $2 + 2 + 0, 2 + 0, 1 + 0, 05$
	his is a greedy approach

make change: artificial example

we have (unlimited number of) coins of value 1, 3, and 4

how to make 6 with a minimum number of coins?

dynamic programming approach:

optimal solution using coins of value 1: $1 + 1 + 1 + 1 + 1 + 1$

optimal solution using coins of value 1 and 3: $3 + 3$

optimal solution using coins of value 1 and 3 and 4: $3 + 3$

optimal solution: $3 + 3$ using dynamic programming

max-subarray problem

assume an array with integers

give the maximum of the sum of the elements of a sub-array

example:

$$[-10, 10, 5, -3, 2, 1]$$
yields 15

max-subarray: algorithm in $O(n^2)$

Algorithm maxSubArray($A, n)$:

$\text{max} := 0$

for $\text{left} := 1 \text{ to } n \text{ do}$

$\text{sum} := 0$

for $\text{right} := \text{left} \text{ to } n \text{ do}$

$\text{sum} := \text{sum} + A[\text{right}]$

if $\text{sum} > \text{max}$ then

$\text{max} := \text{sum}$

return $\text{max}$

we explore all possibilities for left, and given left all possibilities for right
max-subarray: divide and conquer?

improve to $O(n \log n)$ using a divide and conquer approach

it still can be done better

array giving the max sum is
either left of the middle so in $A[low \ldots mid]$
or right of the middle so in $A[mid \ldots high]$
or using the middle so in $A[i \ldots j]$ with $mid \leq i < j \leq high$

algorithm is a bit hairy (not for exam)

max-subarray via divide and conquer in $O(n \log n)$

Algorithm maxSubArray($A, low, high$):

if $low = high$ then
    return $(low, high, A[low])$
else
    mid := $\lfloor (low + high) / 2 \rfloor$
    $(Llow, Lhigh, Lsum) := maxSubArray(A, low, mid)$
    $(Rlow, Rhigh, Rsum) := maxSubArray(A, mid + 1, high)$
    $(Clow, Chigh, Csum) := midMaxSubArray(A, low, mid, high)$
    if $Lsum \geq Rsum$ and $Lsum \geq Csum$ then
        return $(Llow, Lhigh, Lsum)$
    else if $Rsum \geq Lsum$ and $Rsum \geq Csum$ then
        return $(Rlow, Rhigh, Rsum)$
    else
        return $(Clow, Chigh, Csum)$

find max-subarray in the middle

Algorithm midMaxSubArray($A, low, mid, high$):

leftsum := $-\infty$
sum := 0
for $i := mid \text{ downto } low$ do
    sum := sum + $A[i]$
    if sum > leftsum then
        leftsum := sum
        maxleft := $i$
rightsum := $-\infty$
sum := 0
for $j := mid + 1 \text{ to } high$ do
    sum := sum + $A[j]$
    if sum > rightsum then
        rightsum := sum
        maxright := $j$
return $(maxleft, maxright, leftsum + rightsum)$

max-subarray with divide and conquer

worst-case time complexity via a recurrence equation:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n^2) + n & \text{if } n > 1 \end{cases}$$

same as for merge sort, in $\Theta(n \log(n))$

so asymptotically better than brute force which is quadratic
max-subarray: dynamic programming

idea for $B[r]$: maximal sum of subarray ending at index $r$

start:
$B[1] = \max\{A[1], 0\}$

either empty subarray ending at $r$, or continue max-subarray ending at $r - 1$:
$B[r] = \max\{0, B[r - 1] + A[r]\}$

then: max-subarray of the array is max of all the $B[i]$

increase step by step the possibilities

Algorithm maxSubArray($A, n$):

new array $B$
$B[1] := \max(A[1], 0)$
m := $B[1]$

for $r = 2$ to $n$ do

$B[r] := \max(0, B[r - 1] + A[r])$
m := max(m, B[r])

return m

max-subarray: algorithm in $O(n)$

dynamic programming

used for optimization problems such as minimize or maximize

often an efficient version of brute force

solve sub-problems and re-use solutions to sub-problems

memoization and recursion

overview

- make change
- max-subarray
- rod cutting
- knapsack 0/1
- material
rod cutting

problem: rod of $n$ decimeter
table of prices $p_i$ for $i = 1, \ldots, n$ decimeters
determine maximum revenue $r_n$

$2^{n-1}$ possibilities

rod cutting: example

length $n = 4$
table with prices:

<table>
<thead>
<tr>
<th>Length $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price $p_i$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

rod cutting: recursive algorithm

input: array $p[1 \ldots n]$ with prices and int $n$
output: maximum revenue possible for length $m$ ($m \leq n$)

Algorithm rodCuttingRec($p, m$):

if $m = 0$ then
    return 0
$q := -\infty$
for $i := 1$ to $m$ do
    $q := \max(q, p[i] + \text{rodCuttingRec}(p, m - i))$
return $q$

time complexity

recurrence:

$T(0) = 1$
$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$

with induction follows:

$T(n) = 2^n$

intuition: at every possibility we either do or do not cut
see recursion tree for $n = 4$
**Algorithm** rodCuttingDP(p, n):

new array b[0...n]

\[ b[0] := 0 \]

for \( j := 1 \) to \( n \) do

\[ q := -\infty \]

for \( i := 1 \) to \( j \) do

\[ q := \max(q, p[i] + b[j-i]) \]

\[ b[j] := q \]

return \( b[n] \)

---

rod cutting: alternatives

- give not only revenue but also choice for where to cut
- order the pieces in increasing length
- give top-down algorithm using memoization but without recursion

---

overview

- make change
- max-subarray
- rod cutting
- knapsack 0/1
- material
knapsack01: problem

given:
a set $S$ with $n$ items

every item $i$ has weight $w_i$ and benefit $b_i$

maximum total weight $W$

goal:
take items $T \subseteq S$ such that $\sum_{i \in T} b_i$ maximal

under constraint $\sum_{i \in T} w_i \leq W$

naive approach:
consider all possible subsets $T$, this is in $2^n$

knapsack01: idea algorithm

$S_k$ contains elements $1, \ldots, k$
$B[k, w]$ best selection from $S_k$ with total weight $w$
how to find $B[k, w]$?

if $w_k > w$: we cannot take item $k$
so $B[k, w] = B[k - 1, w]$

if $w_k \leq w$: we may take item $i$
$B[k, w] = \max\{B[k - 1, w], B[k - 1, w - w_k] + b_k\}$

knapsack01: algorithm

$S$ consists of $n$ items with $b_i$ and $w_i$; $W$ is the max total weight

**Algorithm 01Knapsack($S, W$):**

```plaintext
new $B[0 \ldots n, 0 \ldots W]
for w := 0 to W do
    $B[0, w] := 0$
for k := 1 to n do
    $B[k, 0] := 0$
    for w := 1 to W do
        if $w_k \leq w$ then
            $B[k, w] := \max(B[k - 1, w], B[k - 1, w - w_k] + b_k)$
        else
            $B[k, w] := B[k - 1, w]$
```

knapsack01: application

- item 1 with $w_1 = 3$ and $b_1 = 9$
- item 2 with $w_2 = 2$ and $b_2 = 5$
- item 3 with $w_3 = 2$ and $b_3 = 5$
- max total weight $W = 4$

- $B[0, 0] = 0$, $B[0, 1] = 0$, $B[0, 2] = 0$, $B[0, 3] = 0$, $B[0, 4] = 0$
- $B[1, 0] = 0$, $B[1, 1] = 0$, $B[1, 2] = 0$, $B[1, 3] = 9$, $B[1, 4] = 9$
**Algorithm 01Knapsack(S, W):**

for $w := 0$ to $W$ do

$B[w] := 0$

for $k := 1$ to $n$ do

for $w := W, \ldots, w_k$ do

if $B[w - w_k] + b_k > B[w]$ then

$B[w] := B[w - w_k] + b_k$

---

**extra material**

- wiki knapsack problem
- wiki NP-completeness