ACMS Seminar Today

- Top Ten Algorithms of the 20th Century
- **The Fast Fourier Transform**
  - **Speaker: Peter Blossey**
- **Smith 205, 3:30-4:20**

CSE 417: Algorithms and Computational Complexity

Dynamic Programming, II

Autumn 2002
Paul Beame

Reading assignment

- Read sections 3.1-3.2 of *The ALGORITHM Design Manual*

Three Steps to Dynamic Programming

- Formulate the answer as a recurrence relation or recursive algorithm
- Show that the number of different parameters in the recursive algorithm is "small"
  - e.g., bounded by a low-degree polynomial
- Specify an order of evaluation for the recurrence so that you already have the partial results ready when you need them.

List partition problem

- Given: a sequence of n positive integers $s_1,\ldots,s_n$ and a positive integer k
- Find: a partition of the list into up to k blocks:
  $s_1,\ldots,s_{i_1}|s_{i_1+1}\ldots s_{i_2}|s_{i_2+1}\ldots s_{i_{k-1}}|s_{i_{k-1}+1}\ldots s_n$
  so that the sum of the numbers in the largest block is as small as possible.
  i.e. find spots for up to k-1 dividers

Greedy approach

- Ideal size would be $P=\sum_{i=1}^{n} s_i/k$
- Greedy: walk along until what you have so far adds up to P then insert a divider
- Problem: it may not be exact (or correct)

  100 200 400 500 900 700 600 800 600
  - sum is 4800 so if k=3 size must be at least 1600.
  - Greedy? Best?

Recursive solution

- Try all possible values for the position of the last divider
- For each position of this last divider
  - there are k-2 other dividers that must divide the list of numbers prior to the last divider as evenly as possible
    - $s_1,\dots,s_{i_1} | s_{i_1+1} \dots s_{i_2} | s_{i_2+1} \dots s_{i_{k-1}} | s_{i_{k-1}+1} \dots s_n$
  - recursive sub-problem of the same type

Recursive idea

- Let M[n,k] the smallest cost (size of largest block) of any partition of the first n #'s into k pieces.
- If best position for last divider lies between the i^{th} and $i+1^{st}$ then

max cost of 1st k-1 blocks
cost of last block

$$M[n,k] = \max\left(M[i,k-1] ,\ \sum_{j=i+1}^{n} s_j \right)$$

- In general

$$M[n,k] = \min_{i<n} \max\left(M[i,k-1] ,\ \sum_{j=i+1}^{n} s_j \right)$$

- Base case(s)?

Time-saving - prefix sums

- Computing the costs of the blocks may be expensive and involved repeated work
- Idea: Pre-compute prefix sums
- Length of block

  $s_{i+1}+\dots+s_j$

  is just

  p[j]-p[i]

- Cost: n additions

$p[1]=s_1$
$p[2]=s_1+s_2$
$p[3]=s_1+s_2+s_3$

...

$p[n]=s_1+s_2+\dots+s_n$

Linear Partition Algorithm

Partition(**S**,**k**):
  p[0]←**0**;
  for **i=1** to **n** do **p[i]** ←**p[i-1]**+s_i
  for **i=1** to **n** do **M[i,1]** ←**p[i]**
  for **j=1** to **k** do **M[1,j]** ← s_1
  for **i=2** to **n** do
    for **j=2** to **k** do
      M[i,j] ← $\min_{pos<i}${max(**M[pos,j-1]**, **p[i]-p[pos]**)}
      D[i,j] ← value of **pos** where min is achieved

Linear Partition Algorithm

Partition(**S**,**k**):
  p[0]←**0**;
  for **i=1** to **n** do **p[i]** ←**p[i-1]**+s_i
  for **i=1** to **n** do **M[i,1]** ←**p[i]**
  for **j=1** to **k** do **M[1,j]** ← s_1
  for **i=2** to **n** do
    for **j=2** to **k** do
      M[i,j]←¥
      for **pos=1** to **i-1** do
        s←max(**M[pos,j-1]**, **p[i]-p[pos]**)
        if **M[i,j]**>**s** then
          M[i,j] ←**s** ; **D[i,j]** ←**pos**

Example:

	1	2	3
100			
200			
400			
500			
900			
700			
600			
800			
600			

Partition(**S**,**k**):
  p[0]←**0**;
  for **i=1** to **n** do **p[i]** ←**p[i-1]**+s_i
  for **i=1** to **n** do **M[i,1]** ←**p[i]**
  for **j=1** to **k** do **M[1,j]** ← s_1
  for **i=2** to **n** do
    for **j=2** to **k** do
      M[i,j] ← $\min_{pos<i}${max(**M[pos,j-1]**, **p[i]-p[pos]**)}
      D[i,j] ← value of **pos** where min is achieved

Example:

	1	2	3
100	100	100	100
200	300		
400	700		
500	1200		
900	2100		
700	2800		
600	3400		
800	4200		
600	4800		

```
Partition(S,k):
    p[0]←0;
    for i=1 to n do p[i] ←p[i-1]+s_i
    for i=1 to n do M[i,1] ←p[i]
    for j=1 to k do M[1,j] ← s_1

    for i=2 to n do
        for j=2 to k do
            M[i,j] ← min_{pos<i}{max(M[pos,j-1],
                                    p[i]-p[pos])}
            D[i,j] ← value of pos where min
                     is achieved
```


Example:

	1	2	3
100	100	100	100
200	300	200	200
400	700	400	400
500	1200	700	500
900	2100	1200	900
700	2800	1600	1200
600	3400	2100	
800	4200	2100	
600	4800	2700	

```
Partition(S,k):
    p[0]←0;
    for i=1 to n do p[i] ←p[i-1]+s_i
    for i=1 to n do M[i,1] ←p[i]
    for j=1 to k do M[1,j] ← s_1

    for i=2 to n do
        for j=2 to k do
            M[i,j] ← min_{pos<i}{max(M[pos,j-1],
                                    p[i]-p[pos])}
            D[i,j] ← value of pos where min
                     is achieved
```


Example:

	1	2	3
100	100	100	100
200	300	200	200
400	700	400	400
500	1200	700	500
900	2100	1200	900
700	2800	1600	1200
600	3400	2100	1300
800	4200	2100	1600
600	4800	2700	2000

```
Partition(S,k):
    p[0]←0;
    for i=1 to n do p[i] ←p[i-1]+s_i
    for i=1 to n do M[i,1] ←p[i]
    for j=1 to k do M[1,j] ← s_1

    for i=2 to n do
        for j=2 to k do
            M[i,j] ← min_{pos<i}{max(M[pos,j-1],
                                    p[i]-p[pos])}
            D[i,j] ← value of pos where min
                     is achieved
```


Longest Increasing Subsequence

- Given a sequence of integers $s_1,\ldots,s_n$ find a **subsequence** $s_{i_1} < s_{i_2} < \ldots < s_{i_k}$ with $i_1 < \ldots < i_k$ so that k is as large as possible.
- e.g. Given **9,5,2,8,7,3,1,6,4** as input,
  - possible increasing subsequence is **5,7**
  - better is **2,3,6** or **2,3,4** (either or which would be a correct output to our problem)

Find recursive algorithm

- Solve sub-problem on $s_1,\ldots,s_{n-1}$ and then try to extend using s_n
- Two cases:
  - s_n is not used
    - answer is the same answer as on $s_1,\ldots,s_{n-1}$
  - s_n is used
    - answer is s_n preceded by the longest increasing subsequence in $s_1,\ldots,s_{n-1}$ that ends in a number smaller than s_n

Refined recursive idea (stronger notion of subproblem)

- Suppose that we knew for each $i<n$ the longest increasing subsequence in $s_1,\ldots,s_n$ that ends in s_i.
  - $i=n-1$ is just the $n-1$ size sub-problem we tried before.
- Now to compute value for $i=n$ find
  - s_n preceded by the maximum over all $i<n$ such that $s_i<s_n$ of the longest increasing subsequence ending in s_i
- First find the best **length** rather than trying to actually compute the sequence itself.

Recurrence

- Let L[i]=length of longest increasing subsequence in $s_1,\ldots,s_n$ that ends in s_i.
- $L[j]=1+\max\{L[i] : i<j \text{ and } s_i<s_j\}$
  (where max of an empty set is 0)
- Length of longest increasing subsequence:
  - $\max\{L[i]: 1\le i \le n\}$

19

Computing the actual sequence

- For each j, we computed
  $L[j]=1+\max\{L[i] : i<j \text{ and } s_i<s_j\}$
  (where max of an empty set is 0)
- Also maintain P[j], the value of the i that achieved that max
  - this will be the index of the predecessor of s_j in a longest increasing subsequence that ends in s_j
  - by following the P[j] values we can reconstruct the whole sequence in linear

20

Longest Increasing Subsequence Algorithm

- **for j=1 to n do**
  - L[j]←1
  - P[j]←0
  - for i=1 to j-1 do
    - if ($s_i<s_j$ & L[i]+1<L[j]) then
      - P[j] ←i
      - L[j] ←L[i]+1
  - endfor
- endfor
- Now find j such that L[j] is largest and walk backwards through P[j] pointers to find the sequence

21

Example

```
for j=1 to n do
    L[j]←1
    P[j]←0
    for i=1 to j-1 do
        if (si<sj & L[i]+1<L[j]) then
            P[j] ←i
            L[j] ←L[i]+1
    endfor
endfor
```

i	1	2	3	4	5	6	7	8	9
s_i									
L[i]									
P[i]									

22

Example

```
for j=1 to n do
    L[j]←1
    P[j]←0
    for i=1 to j-1 do
        if (si<sj & L[i]+1<L[j]) then
            P[j] ←i
            L[j] ←L[i]+1
    endfor
endfor
```

i	1	2	3	4	5	6	7	8	9
s_i	90	50	20	80	70	30	10	60	40
L[i]									
P[i]									

23