Data Structures and Algorithms

Spring 2008-2009

Outline

Outline

Maximum Subsequence Sum Problem

- Given a series of numbers, find the largest continuous sum
- *e.g.* -2 4 -3 5 -2 -1 2 6 -2 1
- Four algorithms solve this, each more efficient than the previous one

Outline

1. Computing the Maximum Subsequence Sum
   - The Problem
   - Four algorithms, one winner

2. Logarithmic Running Time
   - Binary Search

Algorithm 1: Add Up All in Range

- Look at all (start, end) pairs and add up the numbers in between, saving the largest
- There are $\binom{n}{2}$ possible pairings: $O(n^2)$
- In **worst** case the "length" of a pairing will be n
- Average "length" will be $n/2$
- This algorithm will have running time $O(n^3)$

Code can be found here

Algorithm 1: Add Up All in Range (contd.)

- Algorithm comprises three loops, each with non-fixed indices
- Therefore, we should expect the running time to be $O(n^3)$
- Innermost loop does constant work on each iteration so total work done by algorithm is

$$S = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1$$

Looking at rightmost sum, since

$$\sum_{k=i}^{j} 1 = j - i + 1$$

$$S = \sum_{i=1}^{n} \sum_{j=i}^{n} (j - i + 1)$$

Algorithm 1: Add Up All in Range (contd.)

Now,

$$\sum_{j=i}^{n}(j-i+1) = \sum_{k=1}^{n-i+1} k = \frac{(n-i+1)(n-i+2)}{2}$$

Therefore

$$S = \sum_{i=1}^{n} \frac{(n-i+2)(n-i+1)}{2}$$

$$= \frac{1}{2}\sum_{i=1}^{n}(n^2 - in + 2n - in + i^2 - 2i + n - i + 2)$$

$$= \frac{1}{2}\sum_{i=1}^{n}(n^2 + i^2 + i(-2n-3) + 3n + 2)$$

Algorithm 1: Add Up All in Range (contd.)

$$\begin{aligned} S &= \frac{1}{2}\sum_{i=1}^{n}(n^2+3n+2) + \frac{1}{2}\sum_{i=1}^{n} i^2 + \frac{1}{2}(-2n-3)\sum_{i=1}^{n} i \\ &= O(n^3) + O(n^3) + O(n^2) \\ &= O(n^3) \end{aligned}$$

This can be shown more carefully (see P. $\sim$ 53 of *Weiss*) to be

$$\frac{n^3+3n^2+2n}{6}$$

Thus the running time is $O(n^3)$, as predicted.

Algorithm 2: Saving One Loop

- Previous algorithm performed needless recomputations
- Compare work done in computing the subseq. sum `arr[2..5]` and the sum `arr[2..6]`: several additions were repeated
- Can reduce running time to $O(n^2)$ with following modifications (Algorithm 2).
- We can perform a similar analysis to that done previously to show that the running time is $O(n^2)$

Algorithm 3: Divide and Conquer

- Another approach: either the maximum subsequence is in one half of the array or, it is in the other half or, it spans the middle of the array
- To find the maximum sum in the first and second halves use recursion to “divide and conquer” (Latin: *Divide et impera*)
- We can then see if there is a larger subsequence that spans the *middle* of the array
- Spanning the middle gives rise to two pieces but in each piece only one end moves
- Code can be found as Algorithm 3 but note that since solution is recursive it requires an “interface” (driver) function

Algorithm 3: Divide and Conquer (contd.)

Running Time Analysis

- Let $T(n)$ be running time to solve an n-number sequence
- To find the largest subsequence spanning the middle, we will do $O(n)$ work; call it $c \cdot n$
- Then $T(n) = 2T(n/2) + c \cdot n$
- Digression:

  $T(n) = T(n-1) + c$ has solution $T(n) = cn = O(n)$;
  $T(n) = T(n-1) + cn$ has solution $T(n) = cn(n+1)/2 = O(n^2)$

- At each step in *our* algorithm we are halving the size of the problem to be solved
- For $n = 2^k$, can (and will) show that $T(n) = O(n \log n)$

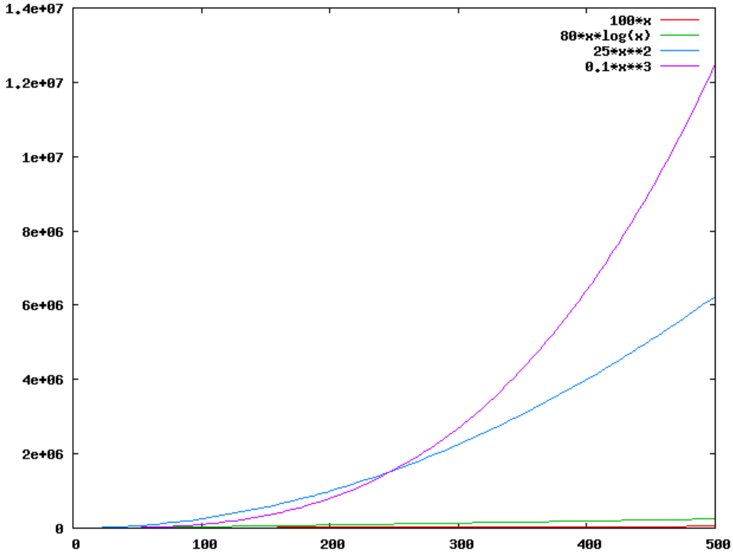
Algorithm 4: A Crucial Observation

- A linear-time algorithm exists for solving MSS.
- The crucial observation is that we would never want to have a negative sum
- As always we remember the best sum we encounter and a running sum
- If running sum ever becomes negative, we may as well reset the starting point to the first positive number
- Algorithm 4

MSS Comparisons

- As is normal with asymptotic analysis (Big-Oh, etc.) we don't give constants on powers of n (because they ultimately don't matter)
- Nonetheless, following picture compares running times of the four algorithms by using some randomly chosen constants
- Comparing $O(n^3)$, $O(n^2)$, $O(n \log n)$ and $O(n)$ in the following picture, $O(n)$ is scraping along the baseline with $O(n \log n)$ just barely diverging from it

MSS Comparisons (contd.)



Outline

Introduction

- If constant time, c, is required to reduce problem size by a constant, k *e.g.*, $T(n) = T(n-k) + c$ are constant, then $T(n) = O(n)$
- An algorithm is $O(\log n)$ if it takes $O(1)$ time to reduce the problem size by a *fraction e.g.*, $T(n) = T(n/2) + 77$
- Binary Search provides one of the fastest search algorithms when data are ordered and indexable (randomly accessible)
- For example, arrays ✓; but not linked lists ✗

Binary Search Code

```
int bin_search(const Atype arr[],
               const Atype& x, const int n)
{
  int lo = 0, hi = n-1;

  while (lo <= hi) {
    int mid = (lo + hi) / 2;
    if (arr[mid] == x) return mid;

    if (arr[mid] < x) lo = mid+1;
    else hi = mid-1;
  }

  return -1;                    // not found
}
```

Analysis

- On each iteration of `while`-loop, size halves
- Assuming $T(1) = c$,

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + c + c \\ &= \quad \vdots \\ &= T(1) + \underbrace{c + c + \cdots + c}_{\lceil \log n \rceil} \\ &= c + c \log n \\ &= O(\log n) \end{aligned}$$