CSE 417: Algorithms and Computational Complexity

Complexity Analysis & Sorting

Autumn 2002
Paul Beame

Reading assignment

- Read Chapter 2 of *The ALGORITHM Design Manual*

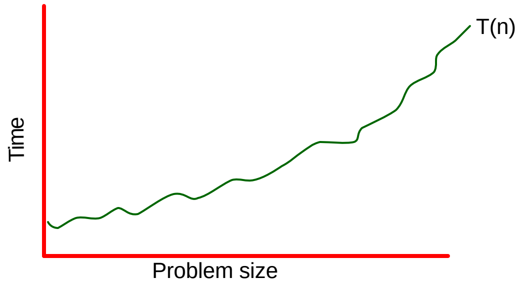
Complexity analysis

- Problem size n
  - **Worst-case complexity**: **max** # steps algorithm takes on any input of size **n**
  - **Best-case complexity: min** # steps algorithm takes on any input of size **n**
  - **Average-case complexity**: **avg** # steps algorithm takes on inputs of size **n**

Complexity

- The complexity of an algorithm associates a number **T(n)**, the best/worst/average-case time the algorithm takes, with each problem size **n**.
- Mathematically,
  - **$T: N^+$ ® R^+**
  - that is **T** is a function that maps positive integers giving problem size to positive real numbers giving number of steps.

Complexity



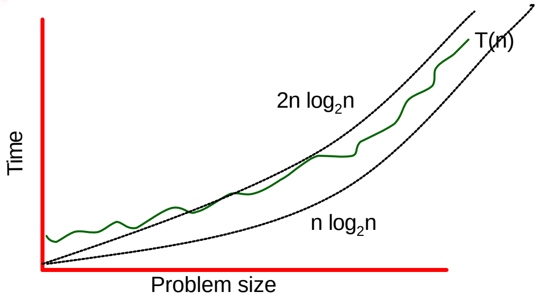
Why Worst-Case Analysis?

- Appropriate for time-critical applications, e.g. avionics
- Unlike Average-Case, no debate about what the right definition is
- Analysis often easier
- Result is often representative of "typical" problem instances
- Of course there are exceptions…

O-notation etc

- Given two functions **f** and **g:N® R**
  - **f(n)** is **O(g(n))** iff there is a constant **c**>0 so that **f(n)** is eventually always £ **c g(n)**
  - **f(n)** is W**(g(n))** iff there is a constant **c**>0 so that **f(n)** is eventually always ³ **c g(n)**
  - **f(n)** is Q**(g(n))** iff there is are constants c_1 and c_2>0 so that eventually always $c_1g(n)$ £ **f(n)** £ $c_2g(n)$

Complexity



Examples

- **$10n^2$-16n+100** is **O(n^2)** also **O(n^3)**
  - $10n^2$-16n+100 ≤ $11n^2$ for all n ≥ 10
- **$10n^2$-16n+100** is W **(n^2)** also W **(n)**
  - $10n^2$-16n+100 ≥ $9n^2$ for all n ≥16
  - Therefore also **$10n^2$-16n+100** is Q**(n^2)**
- **$10n^2$-16n+100** is **not O(n)** also **not** W**(n^3)**
- **Note:** I don't use notation **f(n)=O(g(n))**

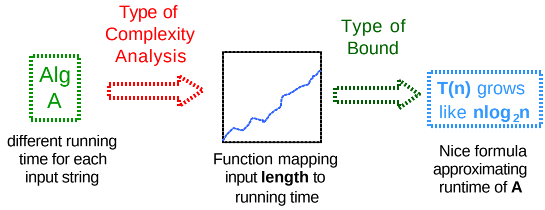
Working with O-W-Q notation

- Claim: For any **a**, **b**>**1** **$\log_a$n** is Q**($\log_b$n)**
  - **$\log_a$n=$\log_a$b** × **$\log_b$n** so letting **c=$\log_a$b** we get that **c$\log_b$n** ≤ **$\log_a$n** ≤ **c$\log_b$n**
- Claim: For any **a** and **b**>**0**, **$(n+a)^b$** is Θ**(n^b)**
  - **$(n+a)^b$** ≤ **$(2n)^b$** for **n** ≥ |**a**| = **2^bn^b** = **cn^b** for **c=2^b** so **$(n+a)^b$** is **O(n^b)**
  - **$(n+a)^b$** ≥ **$(n/2)^b$** for **n** ≥ **2**|**a**| =**$2^{-b}n^b$** =**c'n** for **c'=2^{-b}** so **$(n+a)^b$** is W**(n^b)**

Complexity Analysis

- We have looked at
  - type of complexity analysis
    - worst-case, best-case, average-case
  - types of function bounds
    - **O**, W, Q
- These two considerations are orthogonal to each other
  - one can do any type of function bound with any type of complexity analysis

Complexity analysis overview


Type of Complexity Analysis
Type of Bound
Alg A
T(n) grows like nlog $_2$n
different running time for each input string
Function mapping input **length** to running time
Nice formula approximating runtime of **A**

Usually we represent the function in the middle using a recurrence relation rather than explicitly

General algorithm design paradigm

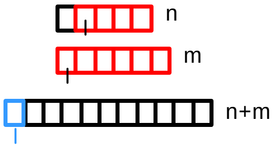
- Find a way to reduce your problem to one or more smaller problems of the same type
- When problems are really small solve them directly

Example

- Mergesort
  - on a problem of size at least 2
    - Sort the first half of the numbers
    - Sort the second half of the numbers
    - Merge the two sorted lists
  - on a problem of size 1 do nothing

Cost of Merge

- Given two lists to merge size n and m
  - Maintain pointer to head of each list
  - Move smaller element to output and advance pointer



Worst case n+m-1 comparisons
Best case min(n,m) comparisons

Recurrence relation for Mergesort

- In total including other operations let’s say each merge costs 3 per element output
- “ceiling” round up
  $T(\mathbf{n})=T(\lceil \mathbf{n}/2 \rceil)+T(\lfloor \mathbf{n}/2 \rfloor)+3\mathbf{n}$ for $\mathbf{n}\geq 2$
- $T(1)=1$ “floor” round down
- Can use this to figure out T for any value of n
  - $T(5)=T(3)+T(2)+3x5$
    $=(T(2)+T(1)+3x3)+(T(1)+T(1)+3x2)+15$
    $=((T(1)+T(1)+3x2)+1+9)+(1+1+6)+15$
    $=8+10+8+15=41$
- $T(\mathbf{n})= 3\mathbf{n}\ \log_2 \mathbf{n}$

Insertion Sort

- For **i**=2 to **n** do
```
  j←i
  while(j>1 & X[ j ] > X[ j-1]) do
        swap X[ j ] and X[ j-1]
```
- i.e., For **i**=2 to **n** do
  Insert **X**[**i**] in the sorted list
  X[**1**],...,**X**[**i-1**]

Recurrence relation for Insertion Sort

- Let $\mathbf{T_n(i)}$ be the **worst case cost** of creating list that has first **i** elements sorted out of **n.**
  - We want to know $\mathbf{T_n(n)}$
- The insertion of **X[i]** makes up to **i-1** comparisons in the worst case
- $\mathbf{T_n(i)=T_n(i-1)+i-1}$ for **i>1**
- $\mathbf{T_n(1)=0}$ since a list of length **1** is always sorted
- Therefore $\mathbf{T_n(n)=n(n-1)/2}$

Solving recurrence relations

- e.g. $T(n)=T(n-1)+f(n)$ for $n \geq 1$
  $T(0)=0$
  - solution is $T(n)=\sum_{i=1}^{n} f(i)$
- Insertion sort: $T_n(i)=T_n(i-1)+i-1$
  - so $T_n(n)=\sum_{i=1}^{n}(i-1)=n(n-1)/2$

Arithmetic Series

- $S= 1 + 2 + 3 + \ldots + (n-1)$
- $S= (n-1)+(n-2)+(n-3)+ \ldots + 1$
- $2S=n + n + n + \ldots + n$ {n-1 terms}
- $2S=n(n-1)$
  - so $S=n(n-1)/2$
- Works generally when $f(i)=a \cdot i+b$ for all i
- Sum = average term size × # of terms

Quicksort

```
Quicksort(X,left,right)
    if left < right
        split=Partition(X, left, right)
        Quicksort(X, left, split-1)
        Quicksort(X, split+1, right)
```

Partition - two finger algorithm

- Partition(**X**, **left**,**right**)

  choose a **random** element to be a **pivot** and pull it out of the array, say at left end

  maintain two fingers starting at each end of the array

  slide them towards each other until you get a pair of elements where right finger has a smaller element and left finger has a bigger one (when compared to pivot)

  swap them and repeat until fingers meet

  put the pivot element where they meet

Partition - two finger algorithm

```
Partition(X,left,right)
    swap X[left], X[random(left, right)]
        pivot ← X[left]; L ← left; R ← right
        while L<R do
            while (X[L] ≤ pivot & L ≤ right) do
                L ← L+1
            while (X[R] > pivot & R ≥ left) do
                R ← R-1
        if L>R then swap X[L],X[R]
    swap X[left],X[R]
    return R
```

In practice

- often choose pivot in fixed way as
  - middle element for small arrays
  - median of 1st, middle, and last for larger arrays
  - median of 3 medians of 3 (9 elements in all) for largest arrays
- four finger algorithm is better
  - also maintain two groups at each end of elements equal to the pivot
    - swap them all into middle at the end of Partition
  - equal elements are bad cases for two fingers

Quicksort Analysis

- Partition does **n-1** comparisons on a list of length **n**
  - pivot is compared to each other element
- If **pivot** is **i**th largest then two sub-problems are of size **i-1** and **n-i**
  - If **pivot** is always in the middle get **T(n)=2T(n/2)+n-1** comparisons
    - **T(n) = n**$\log_2$**n** better than Mergesort
  - If **pivot** is always at the end get **T(n)=T(n-1)+n-1** comparisons
    - **T(n) = n(n-1)/2** like Insertion Sort

Quicksort Analysis Average Case

- Recall
  - Partition does **n-1** comparisons on a list of length **n**
  - If **pivot** is **i**th largest then two sub-problems are of size **i-1** and **n-i**
- Pivot is equally likely to be any one of **1**st through **n**th largest

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}\left(T(i-1) + T(n-i)\right)$$

Quicksort analysis

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}\left(T(i-1) + T(n-i)\right)$$

$$= n - 1 + \frac{2\,T(1) + 2\,T(2) + \ldots + 2\,T(n-1)}{n}$$

$$\therefore nT(n) = n(n-1) + 2\,T(1) + 2\,T(2) + \ldots + 2\,T(n-1)$$

$$\underline{(n+1)T(n+1) = (n+1)n + 2\,T(1) + 2\,T(2) + \ldots + 2\,T(n)}$$

$$\therefore (n+1)T(n+1) - nT(n) = 2\,T(n) + 2n$$

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

$$\therefore \frac{T(n+1)}{n+2} = \frac{T(n)}{n+1} + \frac{2n}{(n+1)(n+2)}$$

Quicksort analysis

$$\text{Let } Q(n) = \frac{T(n)}{n+1}$$

$$\therefore Q(n+1) \le Q(n) + \frac{2}{n+1}$$

$$\therefore Q(n) \le 2\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\right) = 2H_n \approx 2\ln n = 1.38\log_2 n$$

$$(\text{Recall that } \ln n = \int_1^n 1/x\ dx)$$

$$\therefore T(n) \approx 1.38\,n\log_2 n$$

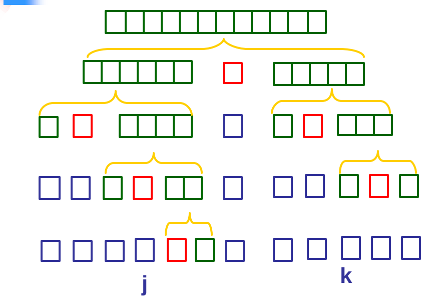
"Gestalt" Analysis of Quicksort

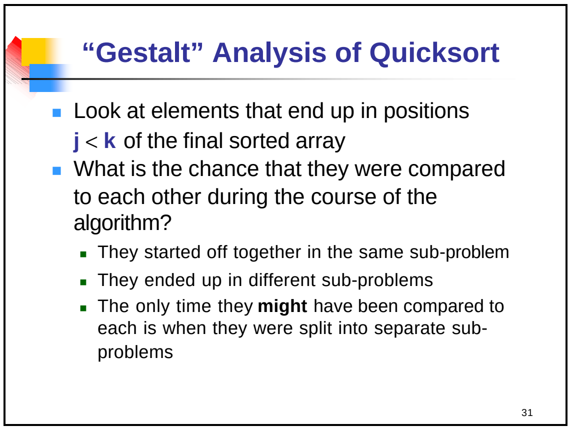
- Look at elements that ended up in positions **j** < **k** of the final sorted array
- The expected # of comparisons in Qsort

  = the expected # of **j** < **k** such that the **j**th and **k**th elements were compared

  = sum$_{j<k}$ Pr[**j**th and **k**th elts were compared]

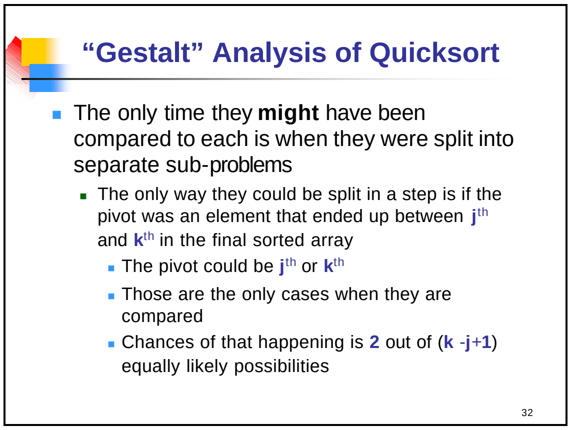
Quicksort execution





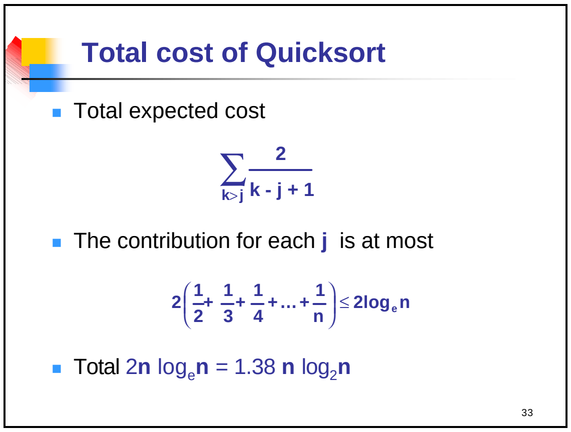
“Gestalt” Analysis of Quicksort

- Look at elements that end up in positions $j < k$ of the final sorted array
- What is the chance that they were compared to each other during the course of the algorithm?
  - They started off together in the same sub-problem
  - They ended up in different sub-problems
  - The only time they **might** have been compared to each is when they were split into separate sub-problems



“Gestalt” Analysis of Quicksort

- The only time they **might** have been compared to each is when they were split into separate sub-problems
  - The only way they could be split in a step is if the pivot was an element that ended up between j^{th} and k^{th} in the final sorted array
    - The pivot could be j^{th} or k^{th}
    - Those are the only cases when they are compared
    - Chances of that happening is 2 out of $(k - j + 1)$ equally likely possibilities



Total cost of Quicksort

- Total expected cost

$$\sum_{k>j} \frac{2}{k - j + 1}$$

- The contribution for each j is at most

$$2\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}\right) \le 2\log_e n$$

- Total $2n \log_e n = 1.38\, n \log_2 n$