Lecture 9: More Sorting

CS 62

Spring 2018

Alexandra Papoutsaki & William Devanny

Assignment 3

- What to do when you want to sort data that cannot fit in memory of your computer?
  - On-disk sorting

- Break data into chunks that will fit in memory, sort chunks, copy into new files: `0.tempfile`, `1.tempfile` , …
- Keep `ArrayList` of files
- Merge files together until one big sorted file.
- Note: You can’t keep file open as both read and write!

Assignment 3 and Lab 3

- Read info on File I/O in Java and file systems in appendix to assignment.
- See on-line Streams cheat sheet

- Lab 3: More complexity/timing (sorting)

Review: Selection Sort

- Goal: sort an array of numbers in non-descending order
- Find smallest element, put it first, sort the rest
- Live code example

Selection Sort correctness

$P(n)$: $\forall n \geq 0$, after running `selectionSort(b,n)`, `b[0..n]` contains the $n + 1$ smallest elements sorted in non-descending order.

Base Case: $i = 0$

`selectionSort` skips the recursive call, finds the minimum element of array `b`, and puts that element in `b[0]`. So the one element array `b[0..0]` contains the 1st smallest element (and is trivially in non-descending order)

Induction Case: $\forall i > 0,\ P(i-1) \Rightarrow P(i)$

Since $i > 0$, the first thing `selectionSort(b,i)` does is recursively call `selectionSort(b,i-1)`. By assumption, when that returns `b[0..i-1]` contains the i smallest elements sorted in non-descending order. `selectionSort` then finds the minimum element in `b[i..]` (which would have to be the $(i+1)\mathrm{th}$ smallest element) and swaps it with the element currently in index i. So `b[0..i]` now contains the $(i+1)$ smallest elements of `b` and, since the first `b[0..i-1]` contains the i smallest, these $i+1$ elements must be sorted in ascending order.

Selection Sort Complexity

To compute the running time of this algorithm, we need to count the number of comparisons in each recursive call

All of the comparison are in `indexOfMin(b,i)`

that makes $n - i$ comparisons

So `selectionSort` makes $n + (n-1) + \dots + 2 + 1$ comparisons

Selection sort takes time $\frac{n(n+1)}{2} = O(n^2)$

FastPower

$fastPower(x, n)$ algorithm to calculate x^n:

- if $n == 0$ then return 1
- if n is even, return $fastPower(x^2, {}^{n}/_{2})$
- if n is odd, return $x * fastPower(x, n - 1)$

FastPower – Proof by strong induction

Base case: $n = 0$

- $x^k = 1$ and $fastPower(x, 0) = 1$
- Assume $fastPower(x, j)$ is x^j for all j ≤ k.
- Show $fastPower(x, k + 1)$ is x^{k+1}
- Case: $k + 1$ is even
  - $fastPower(x, k + 1) = fastPower(x, (k + 1)/2) = (x^2)^{(k+1)/2} = x^{k+1}$
- Case: $k + 1$ is odd
  - $fastPower\,(x, k + 1) = x * fastPower(x, k) = x * x^k = x^{k+1}$

Merge Sort

- Example of Divide & Conquer algorithm
    - Divide array in half
    - Sort each half
    - Merge halves together into completely sorted array

- *Needs extra space (not in-place)*
- Stable: two objects with equal keys appear in the same order in **sorted** output as they appear in the input unsorted array.

MergeSort

```
/**
* MergeSort   Sorts data >= low and < high
* @param list data to be sorted
* @param low start of the data to be sorted
* @param high end of the data to be sorted (exclusive)
*/
private void mergeSort(int[] data, int low, int high){
    if( high-low > 1 ){
        int mid = low + (high-low)/2;
        mergeSort(data, low, mid);
        mergeSort(data, mid, high);
        merge(data, low, mid, high);
    }
}
```

```
/** Merge data >= low and < high into sorted data.
* Data >= low and < mid are in sorted order.
* Data >= mid and < high are also in sorted order
*/
public void merge(int[] data, int low, int mid, int high){
// make temporary array temp of size high-low
int k = 0, i = low, j = mid;
while( i < mid && j < high ){
    if( data[i] <= data[j]){
        temp[k] = data[i];
        i++;
    }else{
        temp[k] = data[j];
        j++;
    }
    k++;
}
// copy over the remaining data on the low to mid side if there is some remaining.
// copy over the remaining data on the mid to high side if there is some remaining.
// Only one of these two while loops should actually execute
// copy the data back from temp to array
```

Example

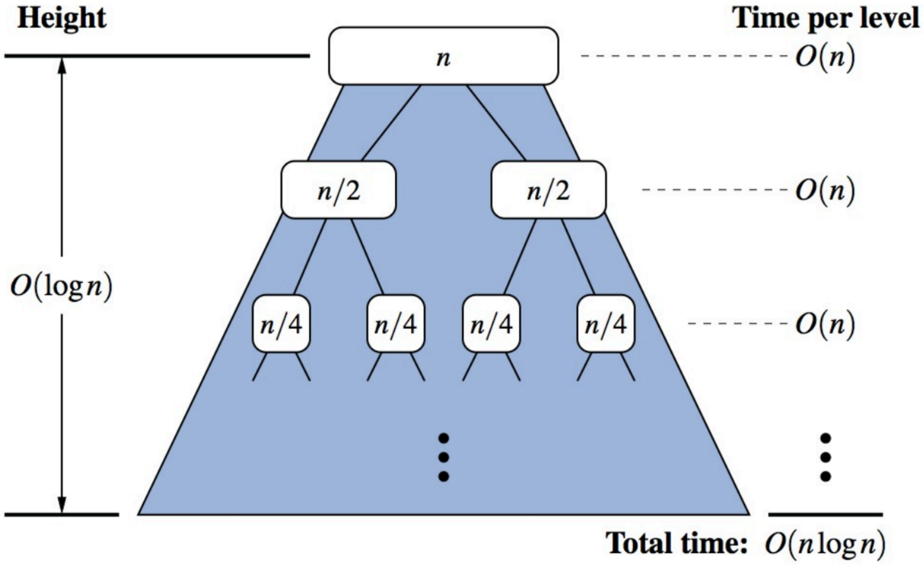
Sort: 85 24 63 45 17 31 96 50 (whiteboard)

Correctness

- $P(n)$: If $high - low = n$ then $mergeSort(data, low, high)$ will result in $data[low\,..\,high]$ being correctly sorted
- For simplicity, assume $merge$ is correct
- Assume $P(k)\ for\ all\ k < n$, show $P(n)$
- If $n = 0$ or 1 then (correctly) do nothing
- Assume $n > 1$
  - Call $mergeSort(data, low, mid)$ and $mergeSort(data, mid + 1, high)$ where $mid = low + (high - low)/2$.
  - Hence $mid - low < n, high - (mid + 1) < n$
  - By induction $data[low..mid]$ and $data[mid + 1\,..\,high]$ now sorted.
  - call $merge(data, low, mid, high)$ and, by assumption on $merge$, $data[low\,..\,high]$ now sorted! Thus $P(n)$ true.

Complexity

- Claim: $mergeSort$ is $O(n \log n)$
    - where $\log$ is base 2
- Merge of two lists of combined size n takes $\leq n - 1$ comparisons.



Complexity

- $P(m)$: if data has 2^m elements then $mergesort$ makes $< m \ * \ 2^m$ total comparisons.
- Assume $P(k) \ for \ all \ k \ < \ 2^m$. Prove $P(m)$
- $P(0), P(1)$ clear. Show $P(m)$
- Sort first half, second half, and then merge
- Each half has size $^{2^m}/_2 = 2^{m-1} < 2^m$, so by induction, each takes $< (m-1) * 2^{m-1}$ comparisons
- Therefore total number of comparisons in $mergesort$
  $$< (m-1) * 2^{m-1} + (m-1) * 2^{m-1} + (2^m - 1)$$
  $$= (m-1) * 2^m + (2^m - 1) = m * 2^m - 1 < m * 2^m$$
- Thus $P(m)$ is true
- If $n = 2^m$ then $mergeSort$ takes $n \log n$ comparisons $(m \ = \ \log n)$.