Mergesort and Recurrences

(Reading: CLRS 2.3, 4.4)

Laura Toma, csci2200, Bowdoin College

We saw a couple of $O(n^2)$ algorithms for sorting. Today we'll see a different approach that runs in $O(n \lg n)$ and uses one of the most powerful techniques for algorithm design, divide-and-conquer.

Outline:

1. Discuss a new sorting algorithm, mergesort.

2. Generalize it to the so-called divide-and-conquer technique.

3. Introduce reccurences as a means to express the running time of recursive algorithms.

4. Show how to solve recurrences by repeated iteration

1 MergeSort

- The idea of mergesort is the following:

- Mergesort:

    - Divide: Divide an array of n elements into two arrays of $n/2$ elements each.
    - Conquer: Sort the two arrays recursively.
    - Combine: Merge the two sorted arrays.

- To implement this, we'll write a recursive procedure $MergeSort(A, p, r)$ which sorts the subarray $A[p..r]$. Initially we'll call $MergeSort(A, 0, n-1)$, that is, $p = 0$ and $r = n - 1$.

- First we'll abstract away from how merge works. Let's assume we have (somebody gives us) a procedure Merge(A, p, q, r) which takes as arguments an array A and three indices p, q, r such that $A[p..q]$ and $A[q+1....r]$ are both sorted, and it merges $A[p..q]$ and $A[q+1....r]$ such that $A[p..r]$ is sorted.

- We can use Merge(A, p, q, r) as a black box to sort $A[p...r]$ as follows:
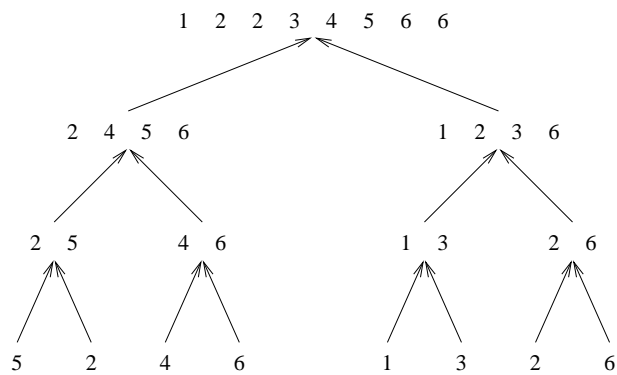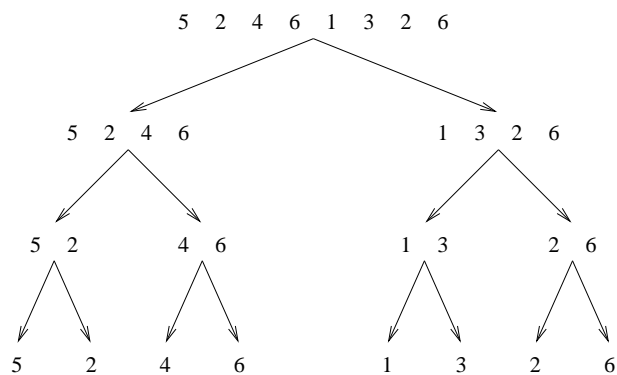
1

```
Merge Sort(A,p,r)
      If p < r then
            q = ⌊(p + r)/2⌋
            MergeSort(A,p,q)
            MergeSort(A,q+1,r)
            Merge(A,p,q,r)
      //else return
```

- Now back to merging: How does Merge(A, p, q, r) work?

    - Imagine merging two sorted piles of cards. The basic idea is to choose the smallest of the two top cards and put it into the output pile.
    - Running time: $\Theta(r - p)$
    - Implementation is a bit messier.

2   Mergesort Example

```
                5   2   4   6   1   3   2   6

          5   2   4   6                   1   3   2   6

     5   2        4   6              1   3        2   6

   5     2      4     6           1     3      2     6
```

```
                1   2   2   3   4   5   6   6

          2   4   5   6                   1   2   3   6

     2   5        4   6              1   3        2   6

   5     2      4     6           1     3      2     6
```

2

3   Mergesort Correctness

1. First, we'll argue that merge is correct, whatever $p \leq q \leq r$ are.

   - Why is merge correct?
   - As you look at the next item to put into the merged output, what has to be true in order for the output to be sorted? Why is this true?

2. Second, we'll prove that $Mergesort()$ is correct when sorting only 1 or 2 elements (the base-case).

3. Third, we'll prove that $Mergesort(A, p, r)$ is correct (for any p, r such that $p \leq r$).

   - Proof idea: Denote the size of the array to be sorted by n. Note that the initial call to Mergesort() the array of size n will call Mergesort() on two arrays of half size, that is, $n/2$. Assuming that these smaller mergesorts are correct, and that merge is correct, and that the base-case is correct, it follows that Mergesort() is correct.
   - Formally, this is called proof by induction on n.
   - Proof:
     - Basecase: Mergesort() is correct when sorting 1 or 2 elements (argue why that's true).
     - Induction hypothesis: Assume that mergesorting any array of size $n/2$ is correct. We'll prove that this implies that mergesorting any array of size n is correct.
     - Proof: mergesorting an array of size n results in two calls to mergesorting arrays of size $n/2$, which, by the induction hypothesis, are correct. Then the results of teh two recursive sorts are merged, and merge, by step 1, is correct. Therefore mergesorting the array of size n is correct.

4   Mergesort Analysis

- To simplify things, let us assume that n is a power of 2, i.e $n = 2^k$ for some k.

- Running time of a recursive algorithm can be analyzed using a **recurrence relation**. Each "divide" step yields two sub-problems of size $n/2$.

- Let $T(n)$ denote the worst-case running time of mergesort on an array of n elements. We have:

$$
\begin{aligned}
T(n) &= c_1 + T(n/2) + T(n/2) + c_2 n \\
&= 2T(n/2) + (c_1 + c_2 n)
\end{aligned}
$$

- Simplified, $T(n) = 2T(n/2) + \Theta(n)$

- We can see that the recurrence has solution $\Theta(n \log_2 n)$ by looking at the **recursion tree**: the total number of levels in the recursion tree is $\log_2 n + 1$ and each level costs linear time (more below).

- Note: If $n \neq 2^k$ the recurrence gets more complicated.

$$T(n) = \begin{cases} \Theta(1) & \text{If } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{If } n > 1 \end{cases}$$

But we are interested in the order of growth, not in the exact answer. So we first solve the simple version (equivalent to assuming that $n = 2^k$ for some constant k, and leaving out base case and constant in Θ). Once we know the solution for the simple version, one needs to solve the original recursion by induction. This step is necessary for a complete proof, but it is rather mechanical, so it is usually skipped.

So even if we are "sloppy" with ceilings and floors, the solution is the same. We usually assume $n = 2^k$ or whatever to avoid complicated cases.

5 Divide-and-conquer

The general idea of mergesort is an instance of a general technique called "divide-and-conquer" which can be applied to any problem P. For e.g. P could be the problem of sorting an array, or finding the smallest element in an array. Divide-and-conquer goes like this:

Divide-and-Conquer

To Solve P:

1. *Divide* P into two smaller problems P_1, P_2.

2. *Conquer* by solving the (smaller) subproblems recursively.

3. *Combine* solutions to P_1, P_2 into solution for P.

The simplest way is to divide into *two* subproblems, as above, but this can be extended to divide into k subproblems.

Analysis of divide-and-conquer algorithms and in general of recursive algorithms leads to recurrences.

6 Solving recurrences

The steps for solving a recurrence relation are the following:

1. Draw the recursion tree to get a feel for how the recursion goes. Sometimes, for easy recurrences, the recursion tree is sufficient to see the bound. This step can be skipped.

2. Iterate and solve the summations to get the final bound.

3. Use induction to prove this bound formally (substitution method).

In this incarnation of the class we will skip the induction step —- generally speaking, once you know induction, this step is pretty mechanical. In this class, solving a recurrence will mean finding a theta-bound for $T(n)$ by iteration.

7 Solving Recurrences by iteration

- Example: Solve $T(n) = 8T(n/2) + n^2$ (with $T(1) = 1$)

$$
\begin{aligned}
T(n) &= n^2 + 8T(n/2) \\
&= n^2 + 8(8T(\frac{n}{2^2}) + (\frac{n}{2})^2) \\
&= n^2 + 8^2 T(\frac{n}{2^2}) + 8(\frac{n^2}{4})) \\
&= n^2 + 2n^2 + 8^2 T(\frac{n}{2^2}) \\
&= n^2 + 2n^2 + 8^2(8T(\frac{n}{2^3}) + (\frac{n}{2^2})^2) \\
&= n^2 + 2n^2 + 8^3 T(\frac{n}{2^3}) + 8^2(\frac{n^2}{4^2})) \\
&= n^2 + 2n^2 + 2^2 n^2 + 8^3 T(\frac{n}{2^3}) \\
&= \ldots \\
&= n^2 + 2n^2 + 2^2 n^2 + 2^3 n^2 + 2^4 n^2 + \ldots
\end{aligned}
$$

  - Recursion depth: How long (how many iterations) it takes until the subproblem has constant size? i times where $\frac{n}{2^i} = 1 \Rightarrow i = \log n$
  - What is the last term? $8^i T(1) = 8^{\log n}$

$$
\begin{aligned}
T(n) &= n^2 + 2n^2 + 2^2 n^2 + 2^3 n^2 + 2^4 n^2 + \ldots + 2^{\log n - 1} n^2 + 8^{\log n} \\
&= \sum_{k=0}^{\log n - 1} 2^k n^2 + 8^{\log n} \\
&= n^2 \sum_{k=0}^{\log n - 1} 2^k + (2^3)^{\log n}
\end{aligned}
$$

- Now $\sum_{k=0}^{\log n - 1} 2^k$ is a geometric sum so we have $\sum_{k=0}^{\log n - 1} 2^k = \Theta(2^{\log n - 1}) = \Theta(n)$
- $(2^3)^{\log n} = (2^{\log n})^3 = n^3$

$$
\begin{aligned}
T(n) &= n^2 \cdot \Theta(n) + n^3 \\
&= \Theta(n^3)
\end{aligned}
$$

8 Other recurrences

Some important/typical bounds on recurrences not covered by master method:

- Logarithmic: $\Theta(\log n)$

    - Recurrence: $T(n) = 1 + T(n/2)$
    - Typical example: Recurse on half the input (and throw half away)
    - Variations: $T(n) = 1 + T(99n/100)$

- Linear: $\Theta(N)$

    - Recurrence: $T(n) = 1 + T(n-1)$
    - Typical example: Single loop
    - Variations: $T(n) = 1 + 2T(n/2), T(n) = n + T(n/2), T(n) = T(n/5) + T(7n/10 + 6) + n$

- Quadratic: $\Theta(n^2)$

    - Recurrence: $T(n) = n + T(n-1)$
    - Typical example: Nested loops

- Exponential: $\Theta(2^n)$

    - Recurrence: $T(n) = 2T(n-1)$