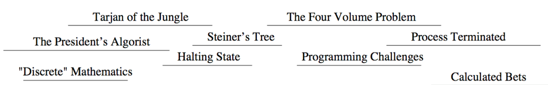
Selecting the Right Jobs

- A movie star wants to the select the maximum number of starring roles such that no two jobs require his presence at the same time.



The Movie Star Scheduling Problem

- **Input**: A set *I* of *n* intervals on the line.
- **Output**: What is the largest subset of mutually non-overlapping intervals that can be selected from *I*?
- Give an algorithm to solve the problem?

Earliest Job First

- Start working as soon as there is work available:
- EarliestJobFirst(I)
  - Accept the earliest starting job *j* from *I* that does not overlap any previously accepted job, and repeat until no more such jobs remain.
- Is this algorithm correct?

First Job to Complete

- Take the job with the earliest completion date:
- OptimalScheduling(I)
  - While($I \neq \emptyset$) do
    - Accept job *j* with the earliest completion date.
    - Delete *j*, and whatever intersects *j* from *I*.
- Is this algorithm correct?

Demonstrating Incorrectness

- Searching for counterexamples is the best way to disprove the correctness of a heuristic.
- Think about all small examples.
- Think about examples with ties on your decision criteria (e.g. pick the nearest point).
- Think about examples with extremes of big and small.

Induction and Recursion

- Failure to find a counterexample to a given algorithm does not mean "it is obvious" that the algorithm is correct.
- Mathematical induction is a very useful method for proving the correctness of recursive algorithms.
- Recursion and induction are the same basic idea: (1) basis case, (2) general assumption, (3) general case.

Best Case, Average Case, Worst Case, Oh My!

- How can we modify almost any algorithm to have a good best-case running time?
- Sorting Example.
- Traveling salesman example.
- A trick used by many!

Best Case

- Too easy to cheat with best case.
- We do not rely it on much.

Average Case

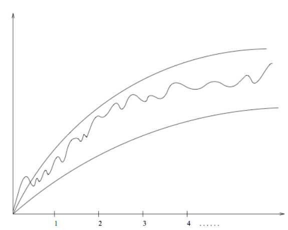
- Usually *very hard* to compute the average running time.
- Very time consuming.

Worst Case

- Fairly easy to analyze.
- Often close to the average running time.
- More informative.

Exact Analysis is Hard

- Best, average, and worst case complexity of an algorithm is a numerical function of the size of the instances.



Exact Analysis is Hard

- It is difficult to work with ***exactly*** because it is typically very complicated.
- It is cleaner and easier to talk about *upper and lower bounds* of the function.
- Remember that we ignore constants.
    - This makes sense since running our algorithm on a machine that is twice as fast will affect the running time by a multiplicative constant of 2, we are going to have to ignore constant factors anyway.

Asymptotic Notation

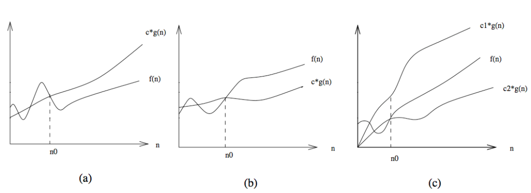
- Asymptotic notation (O, Θ, Ω) are the best that we can practically do to deal with the complexity of functions.

Bounding Functions

- g(n) = O(f(n)) means C x f(n) is an *upper bound on* g(n).
- g(n) = Ω(f(n)) means C x f(n) is a *lower bound on g(n).*
- g(n) = Θ(f(n)) means C_1 x f(n) is an upper bound on g(n) and C_2 x f(n) is a lower bound on g(n).

C, C_1, and C_2 are all constants independent of *n*.

Examples of O, Ω, and Θ



Formal Definitions – Big Oh

- $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $c.g(n)$.
- Think of the equality (=) as meaning *in the set of functions*.

Formal Definitions – Big Omega

Formal Definitions – Big Theta

Logarithms

- It is important to understand deep in your bones what logarithms are and where they come from.
- A logarithm is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $x = \log_b y$.

Logarithms

- Exponential functions, like the amount owed on a n year mortgage at an interest rate of c % per year, are functions which grow distressingly fast, as anyone who has tried to pay off a mortgage knows.
- Thus inverse exponential functions, ie. logarithms, grow refreshingly slowly.

Examples of Logarithmic Functions

- Binary search is an example of an O(lg n) algorithm. After each comparison, we can throw away half the possible number of keys.
- Thus twenty comparisons suffice to find any name in the million-name Manhattan phone book!
- If you have an algorithm which runs in O(lg n) time, take it, because this is blindingly fast even on very large instances.

Another Sorting Algorithm

- What was the running time of insertion sort?
- Can we do better?

Designing Algorithms

- Many ways to design an algorithm:
  - Incremental: *This is what we did with insertion sort. Having sorted the subarray, we insert a single element in its correct position.*
  - Divide and Conquer: *Here the problem is broken up into subproblems that are similar to the original problem but smaller in size. The subproblems are solved recursively then combined to give a solution to the original problem. Merge sort is an example of a divide and conquer algorithm.*

Divide and Conquer

- *Divide the problem into a number of subproblems*
- *Conquer the subproblems by solving them recursively*
- *Combine the subproblem solutions to give a solution to the original problem*

Merge Sort

- Merge Sort is an example of a divide and conquer algorithm

```
MERGE-SORT(A, p, r)
∇ p & r are indices into the array (p < r)
if p < r                    ∇Check for base case
  then q ← ⌊(p + r) / 2⌋     ∇Divide
    MERGE-SORT(A, p, q)      ∇Conquer
    MERGE-SORT(A, q + 1, r)  ∇Conquer
    MERGE(A, p, q, r)        ∇Combine
```


Example

- How would the following array (n=11) be sorted? Since we are sorting the full array, p=1 and r = 11.

4	7	2	6	1	4	7	3	5	2	6

- What would the initial call to MERGE-SORT look like?
- What would the next call to MERGE-SORT look like?
- What would the one after that look like?

The Merge Procedure

- **Input:** Array A and indices p, q, r such that
  - $p \leq q < r$
  - Subarray A[p..q] is sorted and subarray A[q+1..r] is sorted. Neither subarray is empty
- **Output:** The two subarrays are merged into a single sorted subarray in A[p..r]

The Merge Procedure

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1..n1 + 1] and R[1..n2 + 1]
for i ← 1 to n1
  do L[i] ← A[p + i − 1]
for j ← 1 to n2
  do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
  do if L[i] ≤ R[j]
    then A[k] ← L[i]
      i ← i + 1
    else A[k] ← R[j]
      j ← j + 1
```


Example

- A call of MERGE(A, 1, 3, 5) where the array is:

3	5	7	2	6

For Next Time

- Read Chapter 3 from the book.