Prog & Alg I (COMS10002)
Week 7 - Intro to Program Complexity

Dr. Oliver Ray
Department of Computer Science
University of Bristol

Wednesday 12th November, 2014

Fast Integer Exponentiation

- Consider an equivalent definition of exponentiation:

$$x^n = \begin{cases} 1 & \textit{if} \quad n = 0 \\ (x^2)^{n/2} & \textit{if} \quad n \text{ is even} \\ x.x^{n-1} & \textit{if} \quad n \text{ is odd} \end{cases}$$

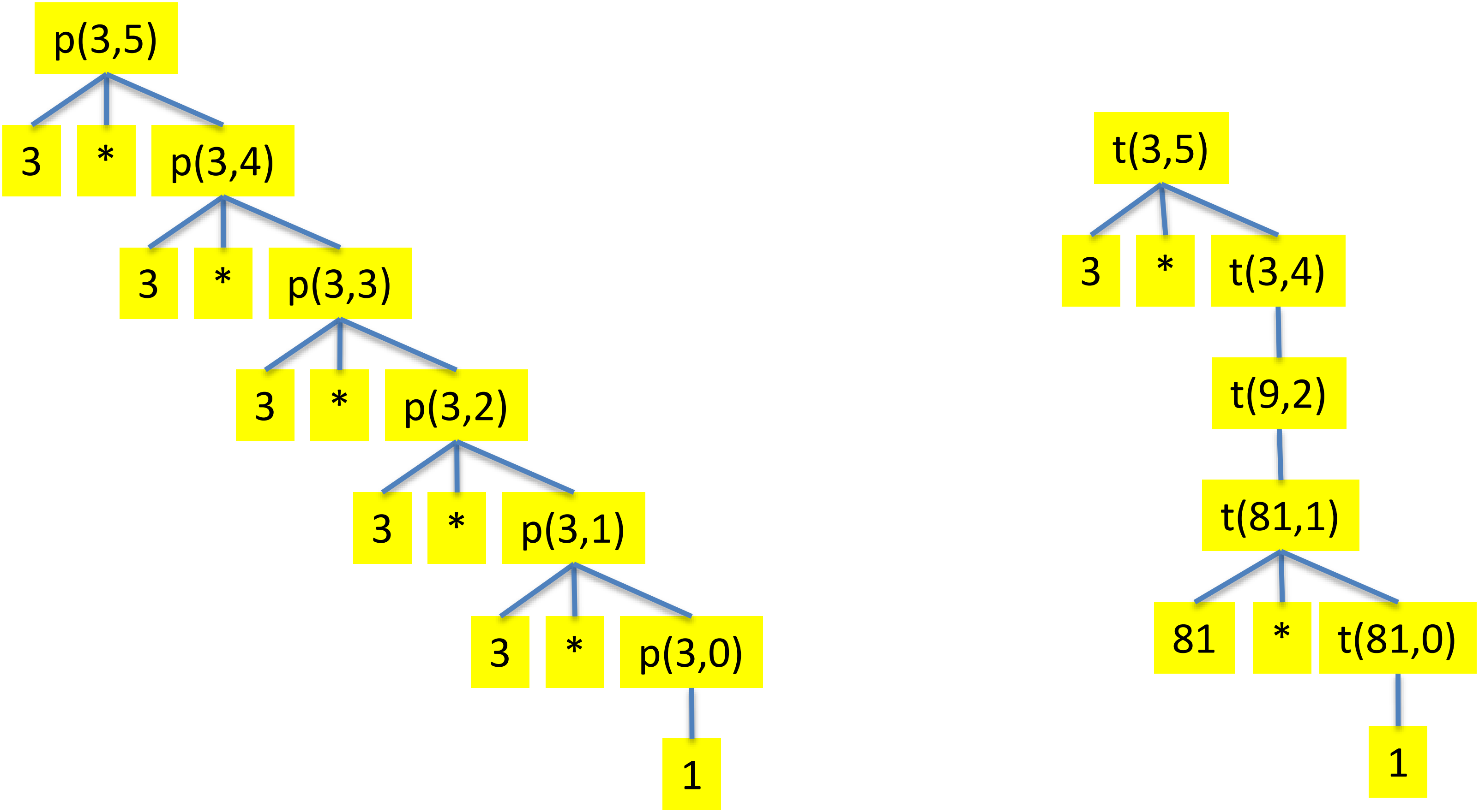
- And its implementation as a recursive function:

```
t(x,n) {
    if (n==0) return 1;
    if ((n&1)==0) return t(x*x,n/2);
    else return x*t(x,n-1);
}
```

Example

```
p(x,n) {
    if (n==0) return 1;
    else return x*p(x,n-1);
}
```

```
t(x,n) {
    if (n==0) return 1;
    if ((n&1)==0) return t(x*x,n/2);
    else return x*t(x,n-1);
}
```



Time Complexity

- We can model the time taken for a function to run on some particular inputs on some particular machine
- We are often satisfied with an approximate model that captures how well the function scales to larger inputs
- We start by defining some basic time constants: e.g.
  - t_c time to compare two values
  - t_a time to perform an arithmetic or logical operation
  - t_f time to call and return from a function
- Then we model the time complexity of a function p() by a corresponding mathematical function $\varphi_p()$
- In practice we simplify all constants to 1, use worst case complexity, and ignore type/stack overflow issues, etc.

Prog & Alg I (COMS10002)
Week 8 – Runtime Complexity

Dr. Oliver Ray
Department of Computer Science
University of Bristol

Wednesday 19th November, 2014

Time Complexity of `p()`

```
p(x,n) {
    if (n==0) return 1;
    else return x*p(x,n-1);
}
```

$$\varphi_p(x,n) = \begin{cases} t_f + t_c & if \quad n = 0 \\ t_f + t_c + 2t_a + \varphi_p(x,n-1) & if \quad n > 0 \end{cases}$$

$$\varphi_p(x,n) = \begin{cases} 2 & if \quad n = 0 \\ 4 + \varphi_p(x,n-1) & if \quad n > 0 \end{cases}$$

- Note: here the complexity is independent of *x*
- Exercise: derive corresponding functions for `q` and `r`

Exact Bound

```
p(x,n) {
    if (n==0) return 1;
    else return x*p(x,n-1);
}
```

$$\varphi_p(x,0) = 2$$

$$\varphi_p(x,1) = 4 + 2 = 6$$

$$\varphi_p(x,2) = 4 + 4 + 2 = 10$$

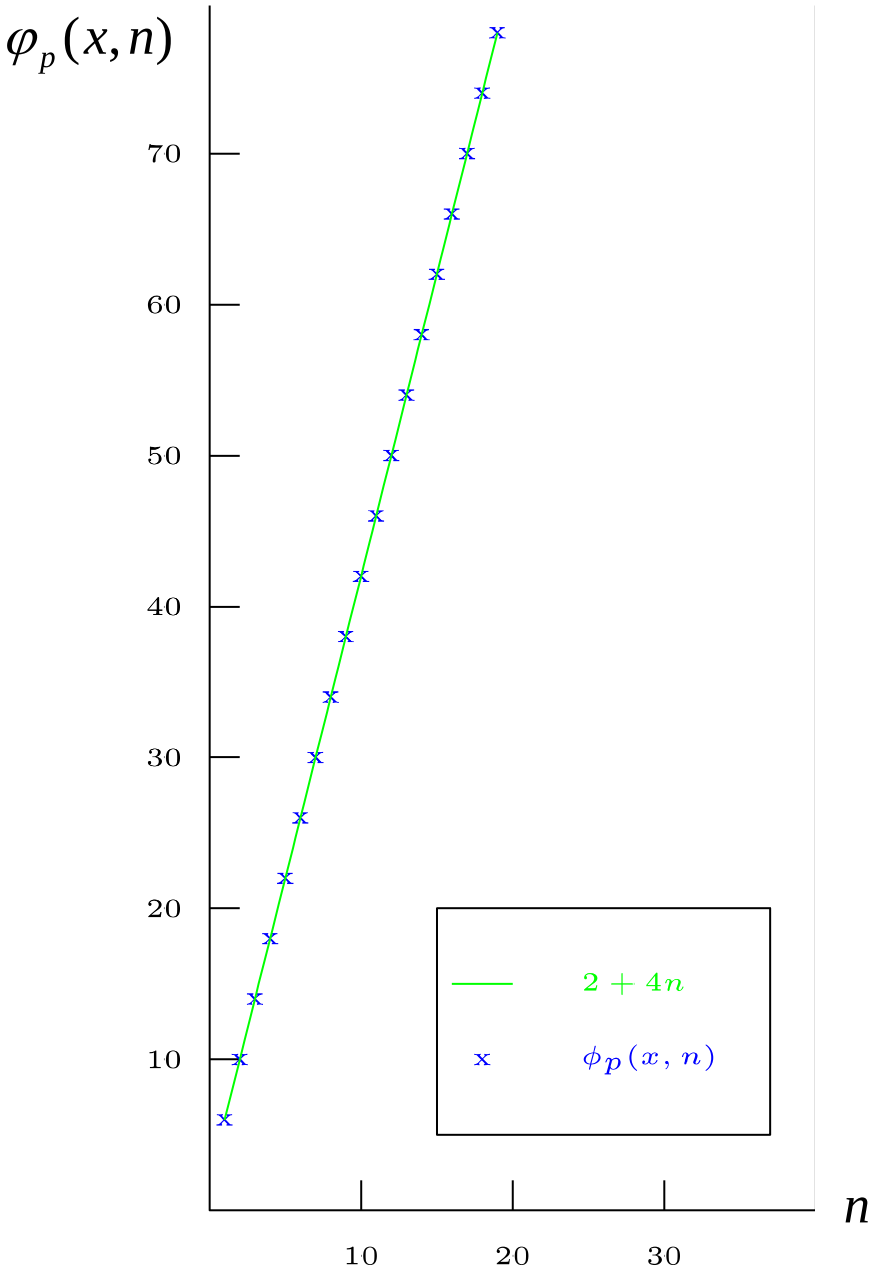
$$\varphi_p(x,3) = 4 + 4 + 4 + 2 = 14$$

$$\varphi_p(x,4) = 4 + 4 + 4 + 4 + 2 = 18$$

$\cdots$

$$\varphi_p(x,n) = 2 + 4n \qquad \text{for all } x \in \mathrm{Z}, n \in \mathrm{N}$$

Linear Growth



Time Complexity of `t()`

```
t(x,n) {
    if (n==0) return 1;
    if ((n&1)==0) return t(x*x,n/2);
    else return x*t(x,n-1);
}
```

$$\varphi_t(x,n)=\begin{cases} t_f+t_c & if \quad n=0 \\ t_f+2t_c+3t_a+\varphi_t(x^2,n/2) & if \quad n \text{ is even} \\ t_f+2t_c+3t_a+\varphi_t(x,n-1) & if \quad n \text{ is odd} \end{cases}$$

$$\varphi_t(x,n)=\begin{cases} 2 & if \quad n=0 \\ 6+\varphi_t(x^2,n/2) & if \quad n \text{ is even} \\ 6+\varphi_t(x,n-1) & if \quad n \text{ is odd} \end{cases}$$

Upper Bound

```
t(x,n) {
   if (n==0) return 1;
   if ((n&1)==0) return t(x*x,n/2);
   else return x*t(x,n-1);
}
```

1 time — *cost 2*

i + j -1 times — *cost 6*

j times — cost 6

where *i* and *j* denote the number of 0's and 1's in the binary representation (ignoring padding) of any **non-zero** *n*

$$i + j - 1 = \lfloor \log_2 n \rfloor = k$$

1 ≤ j ≤ k+1

j=1	(best case)
j=1+k	**(worst case)**
j=1+k/2	(on average)

$$\varphi_t(x,0) = 2$$

$$\varphi_t(x,1) = 6 + 2 = 8$$

$$\varphi_t(x,2) = 6 + 6 + 2 = 14$$

$$\varphi_t(x,3) = 6 + 6 + 6 + 2 = 20$$

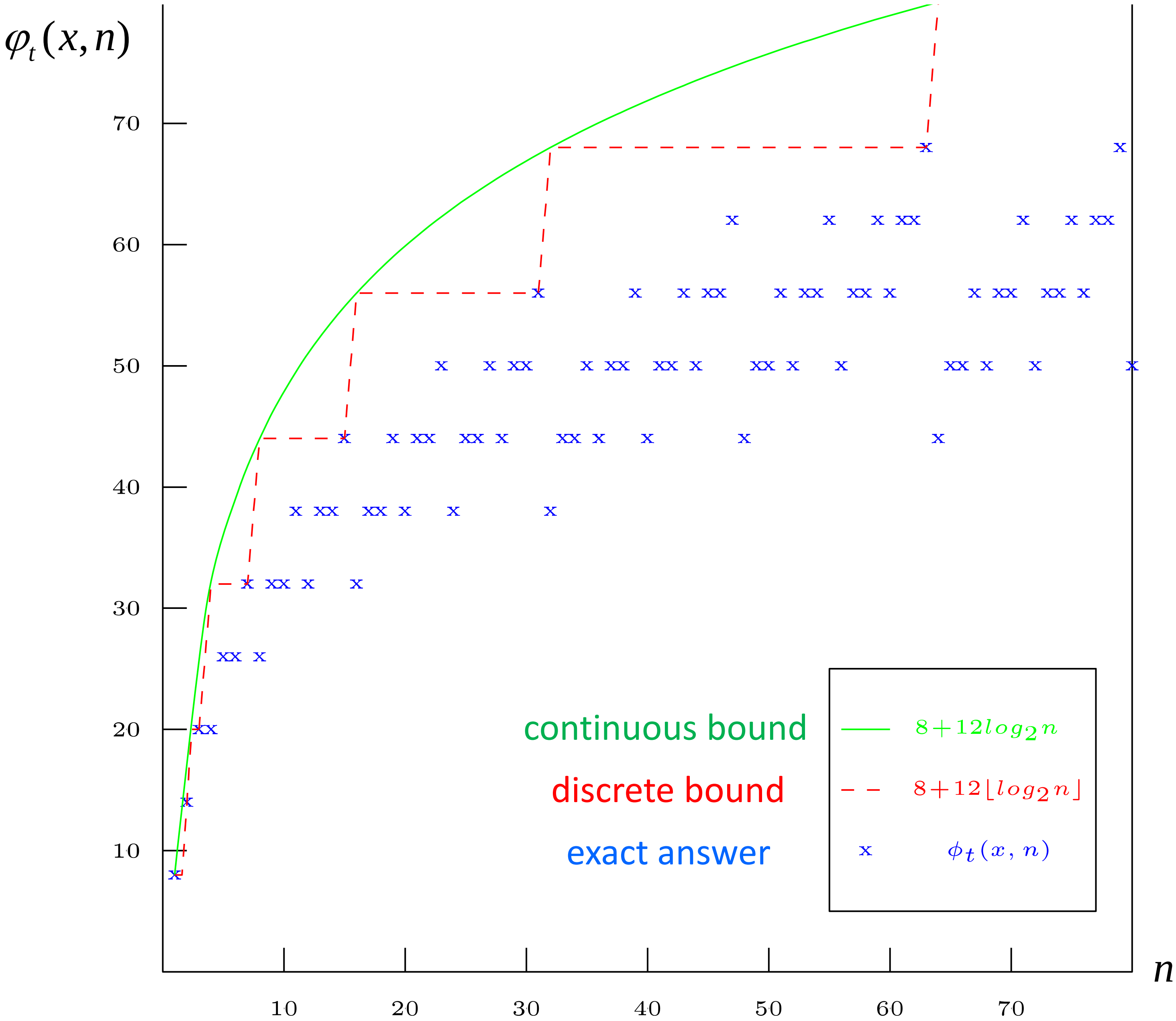
$$\varphi_t(x,4) = 6 + 6 + 6 + 2 = 20$$

…

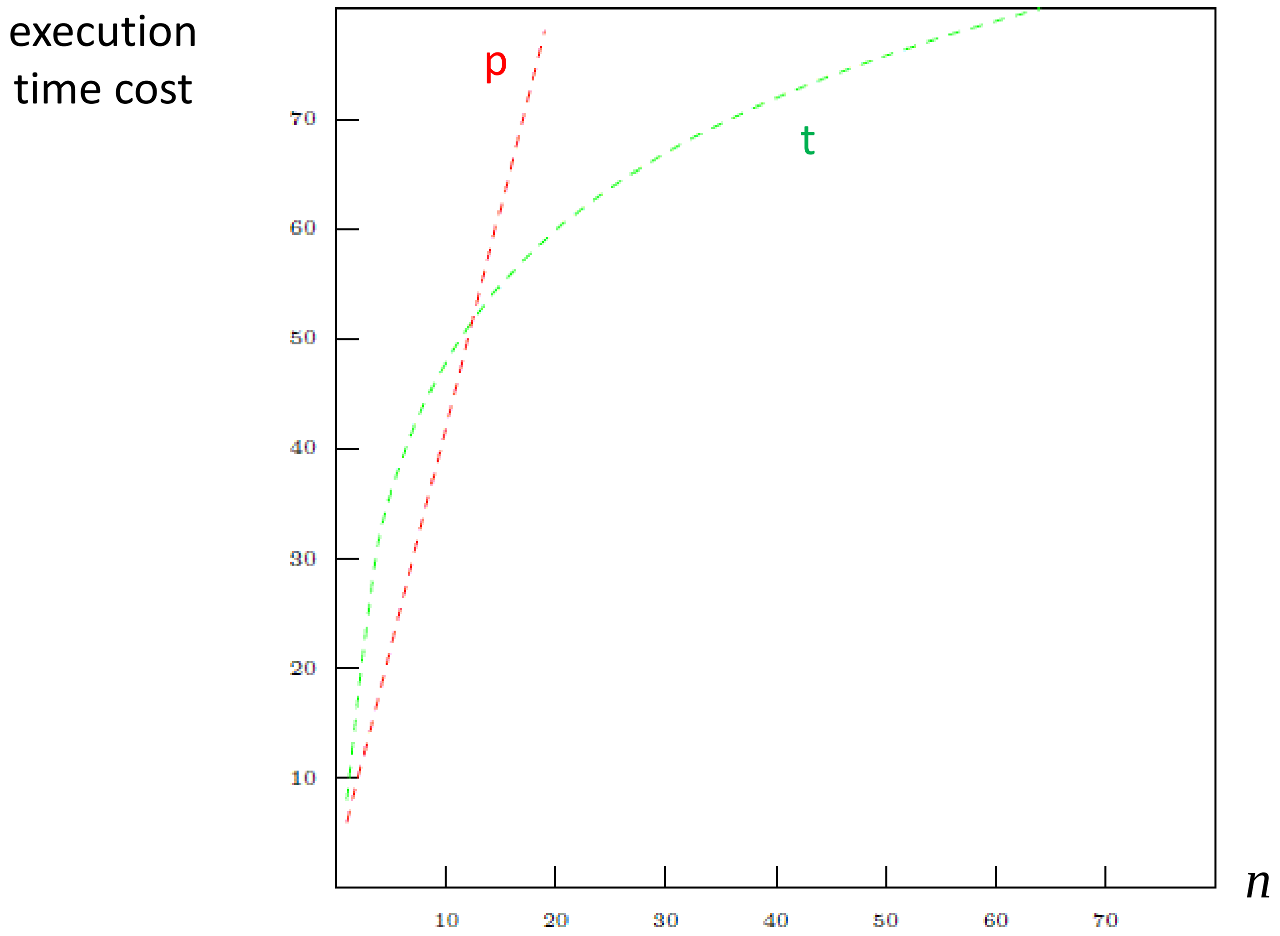
$$\varphi_t(x,n) \le 2 + 6k + 6(k+1) = 8 + 12\lfloor \log_2 n \rfloor \le 8 + 12\log_2 n$$

for all $n > 0$

Logarithmic Growth



Which is better?



- p is **slightly** better than t for very **small** values of n
- but t is **significantly** better than p for **all** other values of n

Practical Consequence

- Running t on an laptop will turn out to be *faster* than running p on a supercomputer *for large enough* n

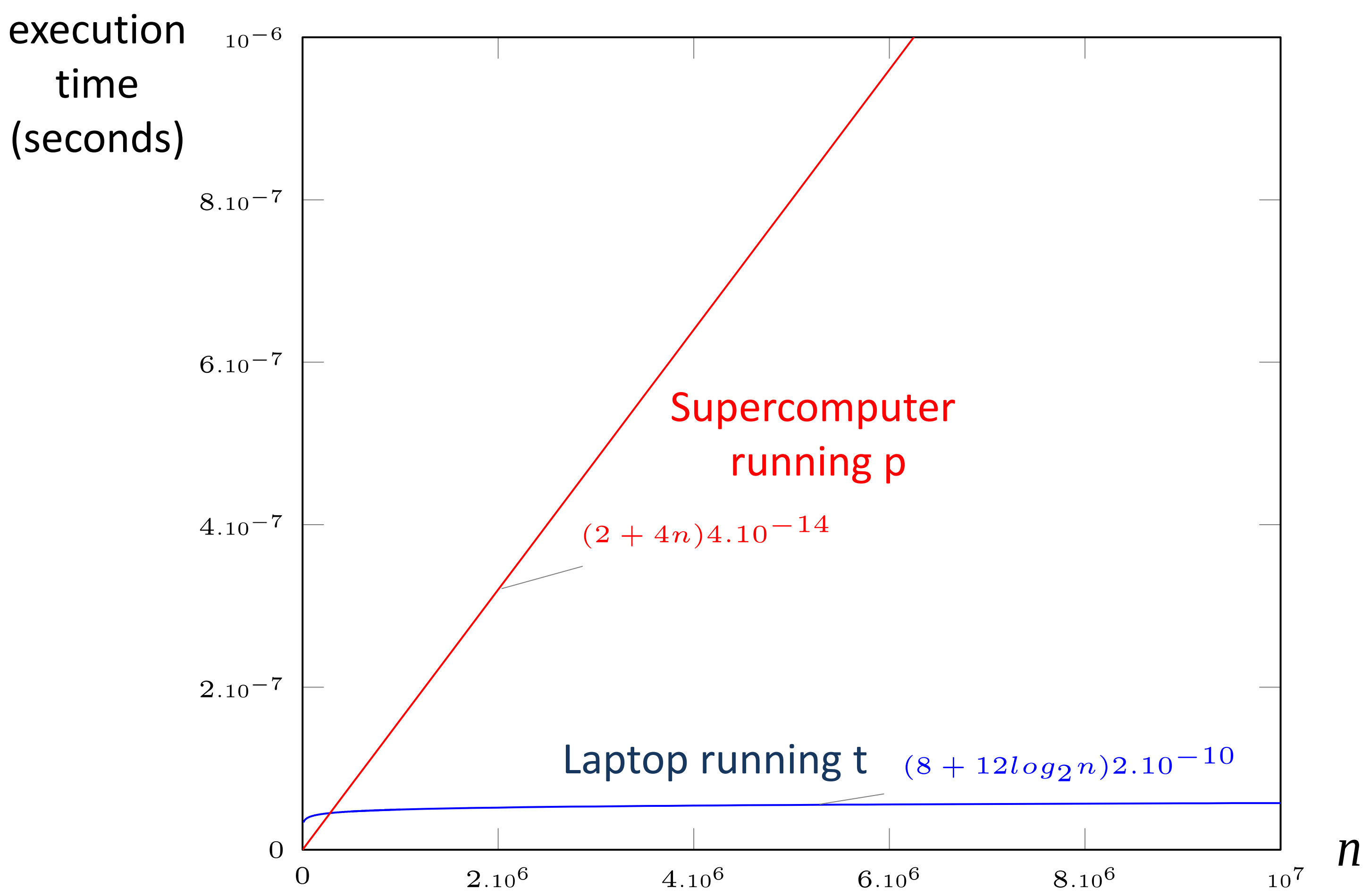
MacBook(≈5 GFlops/sec)

time ≈ $(8+12\log_2 n)2.10^{-10}$ sec

BlueCrystal (≈25 TFlops/sec)

time ≈ $(2+4n)\ 4.10^{-14}$ sec

(Estimated) Execution Time



- The constants t_c,t_a,t_f and the computer speeds don't alter this
- It is the difference between log and linear scaling that matters

Exercise

- Derive a lower bound for **best** case performance of t
- Characterise the **average** case performance of t
- Discuss how realistic our assumptions are

Prog & Alg I (COMS10002)
Week 9 –Complexity Classes

Dr. Oliver Ray
Department of Computer Science
University of Bristol

Wednesday 26th November, 2014

Big-Oh and the “Order” of a Function

- A function $f(n)$ is said to be order $g(n)$ iff $|f(n)| \leq c|g(n)|$ for all $n \geq k$ where c and k are finite positive constants
- We denote this fact by writing $f(n)=O(g(n))$ and we say that “f is Big-Oh of g”
- If f and g are positive functions (as they usually are in the study of runtimes!) we don’t need to take moduli
- Intuitively this means that for all sufficiently large n ($\geq k$) f is less than some constant multiple (c) of g

Order of p()

- Thus we can say function p runs in order n

  - take $f(n)=2+4n$ $g(n)=n$
  - guess $c=5$
  - solve $2+4n \le 5n$

    $2 \le 5n-4n$

    $2 \le n$
  - thus $k = 2$

Example

- Thus we can say function t runs in order $\log_2 n$
  - take $f(n)=8+12\log_2 n$ $g(n)=\log_2 n$
  - guess $c=20$
  - solve $8+12\log_2 n \le 20\log_2 n$

    $8 \le 8\log_2 n$

    $1 \le \log_2 n$

    $2^1 \le 2^{\log 2n}$

    $2 \le n$
  - thus $k=2$

Alternative Characterisation of Big-Oh

- It is sometimes hard or inconvenient having to work out the constants *c* and *k* in the above definition of Big-Oh
- We can obtain an alternative characterisation of Big-Oh using the fact that $f(n)=O(g(n))$ if the following holds:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}<\infty$$

i.e. the ratio of the two functions tends to a finite limit

Example

- n^2+2n+4 is order n^2

$$\lim_{n\to\infty}\frac{n^2+2n+4}{n^2}=\lim_{n\to\infty}\left(\frac{n^2}{n^2}+\frac{2n}{n^2}+\frac{4}{n^2}\right)$$

$$=\lim_{n\to\infty}\left(1+\frac{2}{n}+\frac{4}{n^2}\right)$$

$$=1$$

$$<\infty$$

Comparison of Runtime Behaviors