Tidskomplexitet

725G97: DALG

Magnus Nielsen

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Funktionsbegrepp

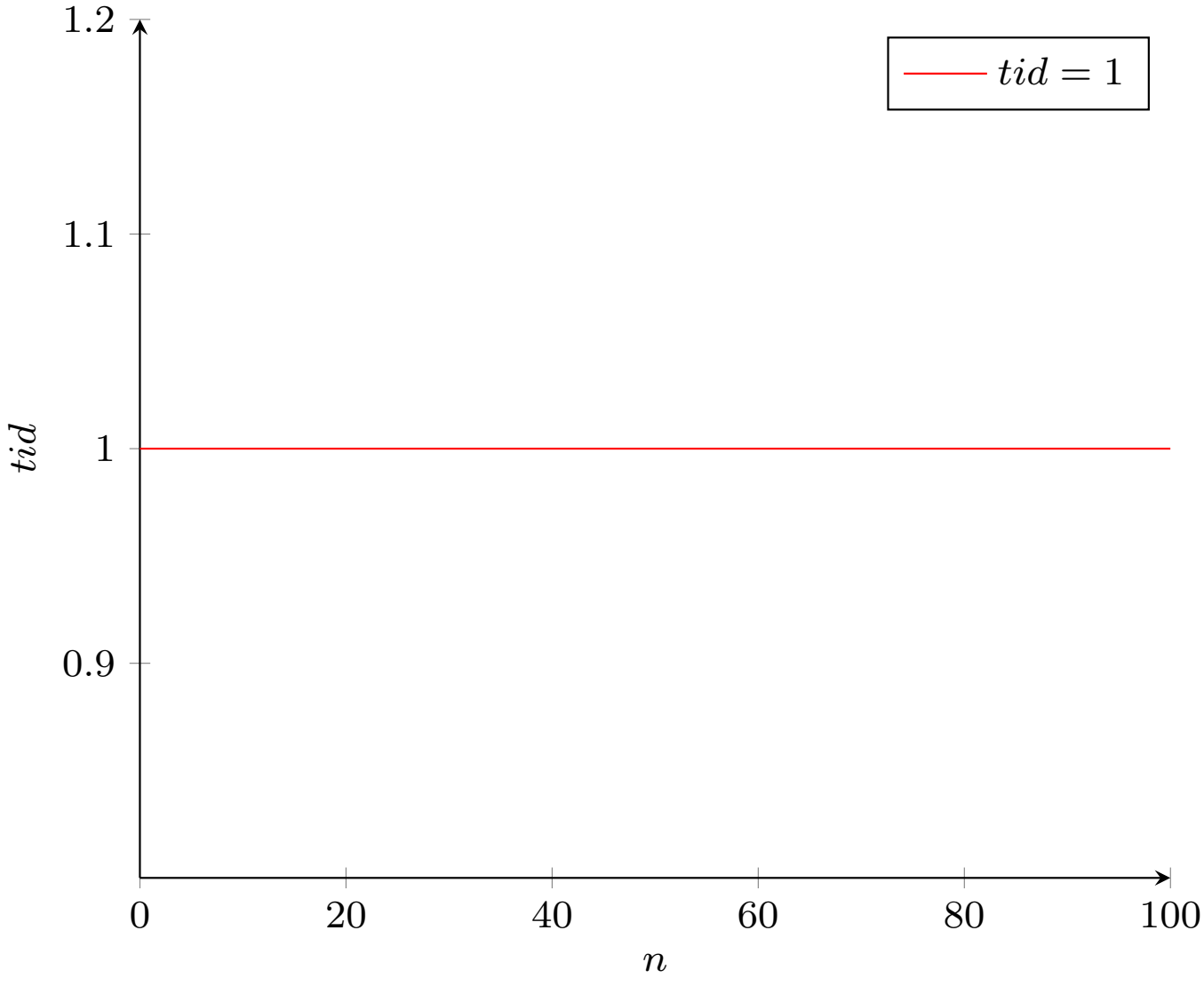
Det finns oändligt många funktionsbegrepp som *kan* användas för att beskriva tidskomplexitet. I kurser som denna använder vi ett fåtal, de som är vanligast i praktiken och generellt enklast att identifiera.

- $\mathcal{O}(1)$ - Konstant tillväxt
- $\mathcal{O}(\log(n))$ - Logaritmisk tillväxt
- $\mathcal{O}(n)$ - Linjär tillväxt
- $\mathcal{O}(n^2)$ - Kvadratisk tillväxt
- $\mathcal{O}(n^x)$ - Större potentsfunktioner (n upphöjt till något x > 2)

Mindre vanliga (generellt rekursiva):

- $\mathcal{O}(2^n)$ - Exponentiell tillväxt
- $\mathcal{O}(n!)$ - n-fakultet tillväxt.

Konstant tillväxt



Exempel på funktion

```
int plusOne(int n) {
  // Vi har inga iterationer, rekursiva anrop...
  return n + 1;
}
```

Intuition:

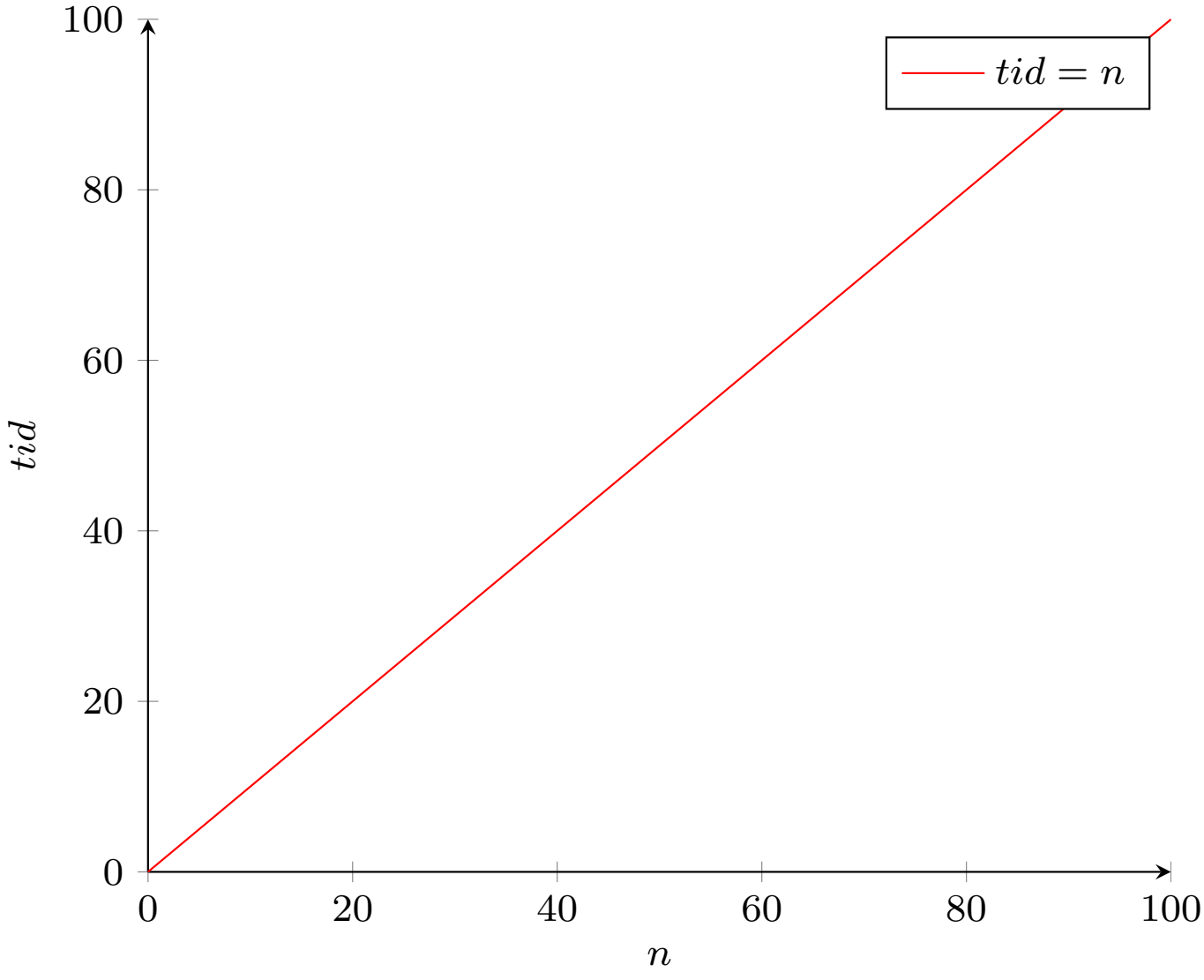
Exempel på funktion

```
int plusOne(int n) {
  // Vi har inga iterationer, rekursiva anrop...
  return n + 1;
}
```

Intuition: Oavsett hur stort n är kommer det inte påverka körtiden på funktionen.

Alltså:
Alla funktioner vars körtid inte påverkas av storleken på den data de arbetar med har en tidskomplexitet $f(n) \in \mathcal{O}(1)$.

Linjär tillväxt



Exempel på funktion

```
int addUpTo(int n) {
  int result = 0;

  // Linjär loop: Vi tar ungefär ett steg i taget
  // från ungefär ingenting till ungefär n.
  for (int i = 1; i < n; i++) {
    // Innehållet i loopen tar konstant tid
    result += i;
  }
  return result;
}
```

Intuition:

Exempel på funktion

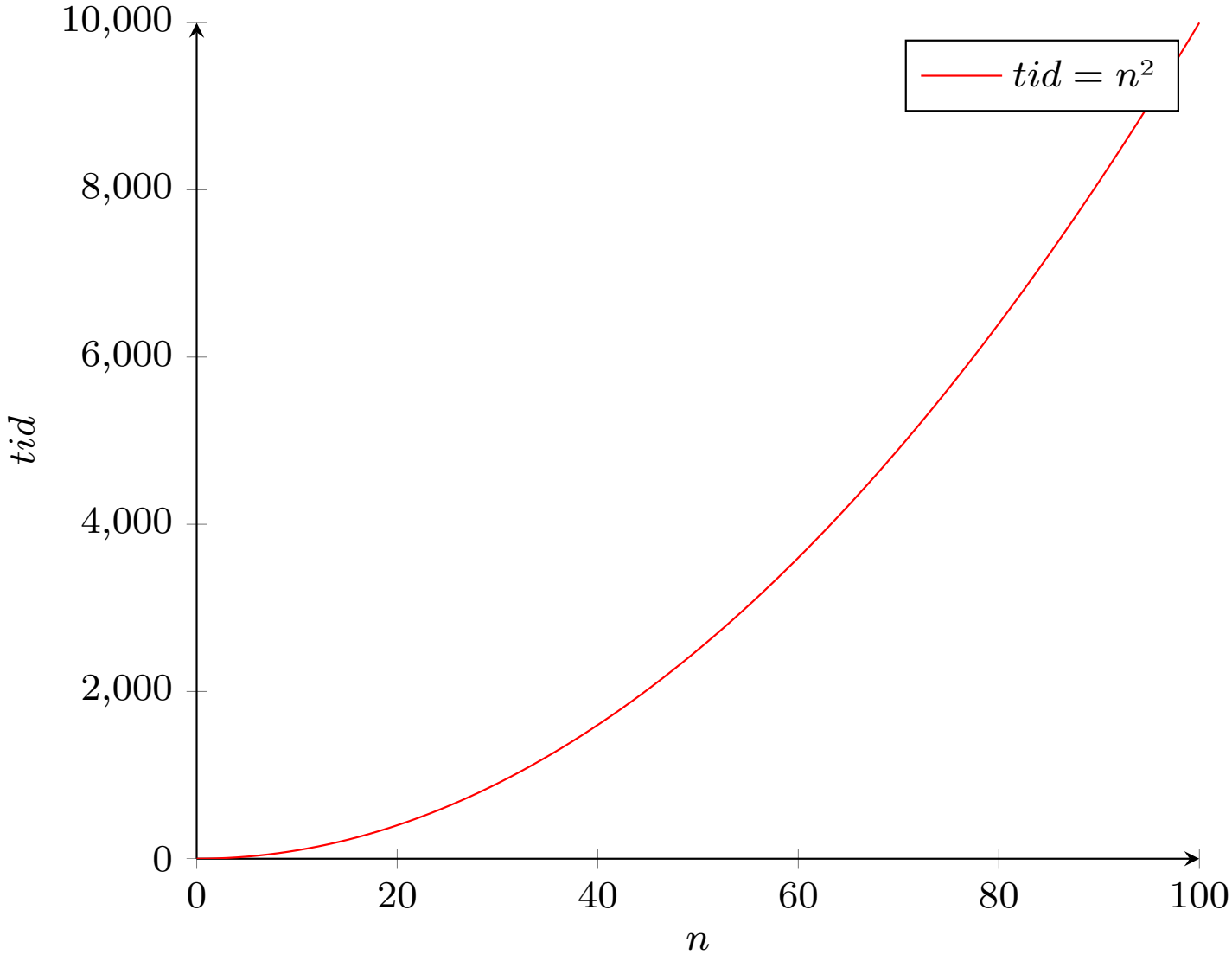
```
int addUpTo(int n) {
  int result = 0;

  // Linjär loop: Vi tar ungefär ett steg i taget
  // från ungefär ingenting till ungefär n.
  for (int i = 1; i < n; i++) {
    // Innehållet i loopen tar konstant tid
    result += i;
  }
  return result;
}
```

Intuition: Körtiden kommer att växa linjärt med storleken på indata.

Alltså:
Om det tar 15 tidsenheter att exekvera funktionen då $n = 3$, bör det ta ungefär 25 tidsenheter att exekvera densamma då $n = 5$.
Samma intuition bör gälla om vi tar 2, 3 eller 4 steg i loopen istället för 1: funktionen kommer ju att "skala upp" likvärdigt på riktigt stora n.

Kvadratisk tillväxt



Exempel på funktioner

```
int countOperationsSquareTime(int n) {
  int result = 0;

  // Vi tar ungefär n^2 steg, ett i taget
  for (int i = 0; i < n*n; i++) {
    result += 1;
  }
  return result;
}

int countOperationsSquareTime2(int n) {
  int result = 0;

  // Nästlade linjära loopar
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      result += 1;
    }
  }
  return result;
}
```

Slutsatser

Intuition:

Slutsatser

Intuition: Nästlade linjära loopar har kvadratisk komplexitet.
Linjära loopar som tar kvadratiskt (n^2) antal steg.

Körtiden kommer att växa kvadratiskt med storleken på indata.

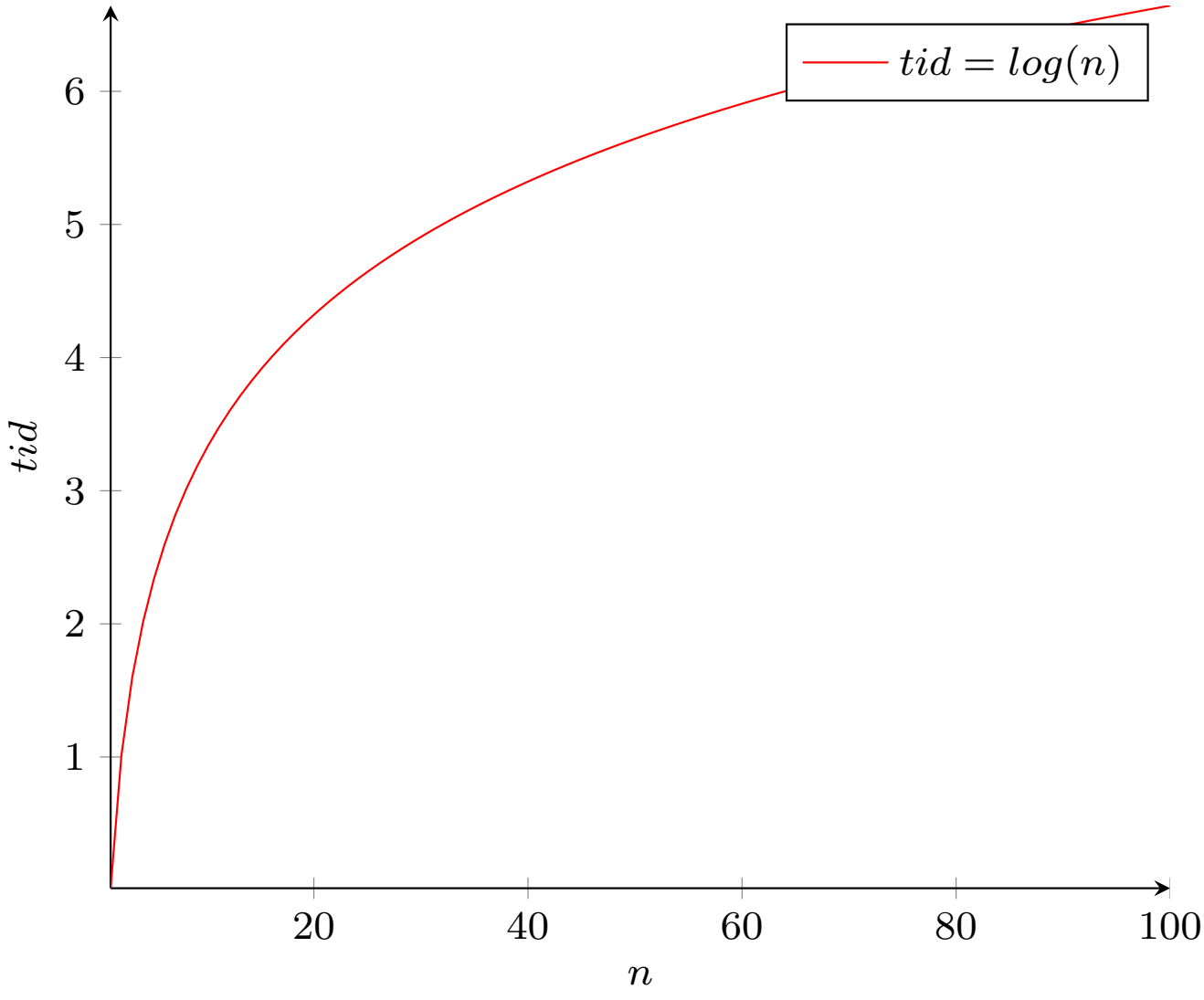
Alltså:
Om det tar 100 tidsenheter att exekvera funktionen då n $= 10$, bör det ta ungefär 121 tidsenheter att exekvera densamma då n $= 11$.
Samma intuition (precis som linjärt) bör gälla om vi tar 2, 3 eller 4 steg i loopen istället för 1: funktionen kommer ju att "skala upp" likvärdigt på riktigt stora n.

n^x: Samma tänk gäller oavsett vad x råkar vara. Ex:

```
for (int i = 0; i < n*n*n; i++) {
```

$n * n * n$ och $i{+}{+} \implies \mathcal{O}(n^3)$.

Logaritmisk tillväxt



Exempel 1 på funktioner

```
int countOperationsLogTime(int n) {
  result = 0;
  // Vi tar dubbelt så stort steg mot målet
  // för varje iteration
  for (int i = 1; i < n; i = i * 2) {
    result += 1;
  }
}
```

Vi kan se att vår loop-räknare kommer att anta värdena 1, 2, 4, 8, 16, 32 ... vilket råkar sammanfalla mycket väl med tvåpotenserna.

Exempel 2 på funktioner

```
boolean binarySearch(int arr[], int key){
  int mid;
  int first = 0;
  int last = arr.length-1;
  while( first <= last ){
    mid = (first + last)/2;
    if ( arr[mid] < key ){
      first = mid + 1;
    }else if ( arr[mid] == key ){
      return true; // Vi hittade nyckeln!
    }else{
      last = mid - 1;
    }
  }
  return false; // Vi hittade inte nyckeln
}
```

Slutsatser

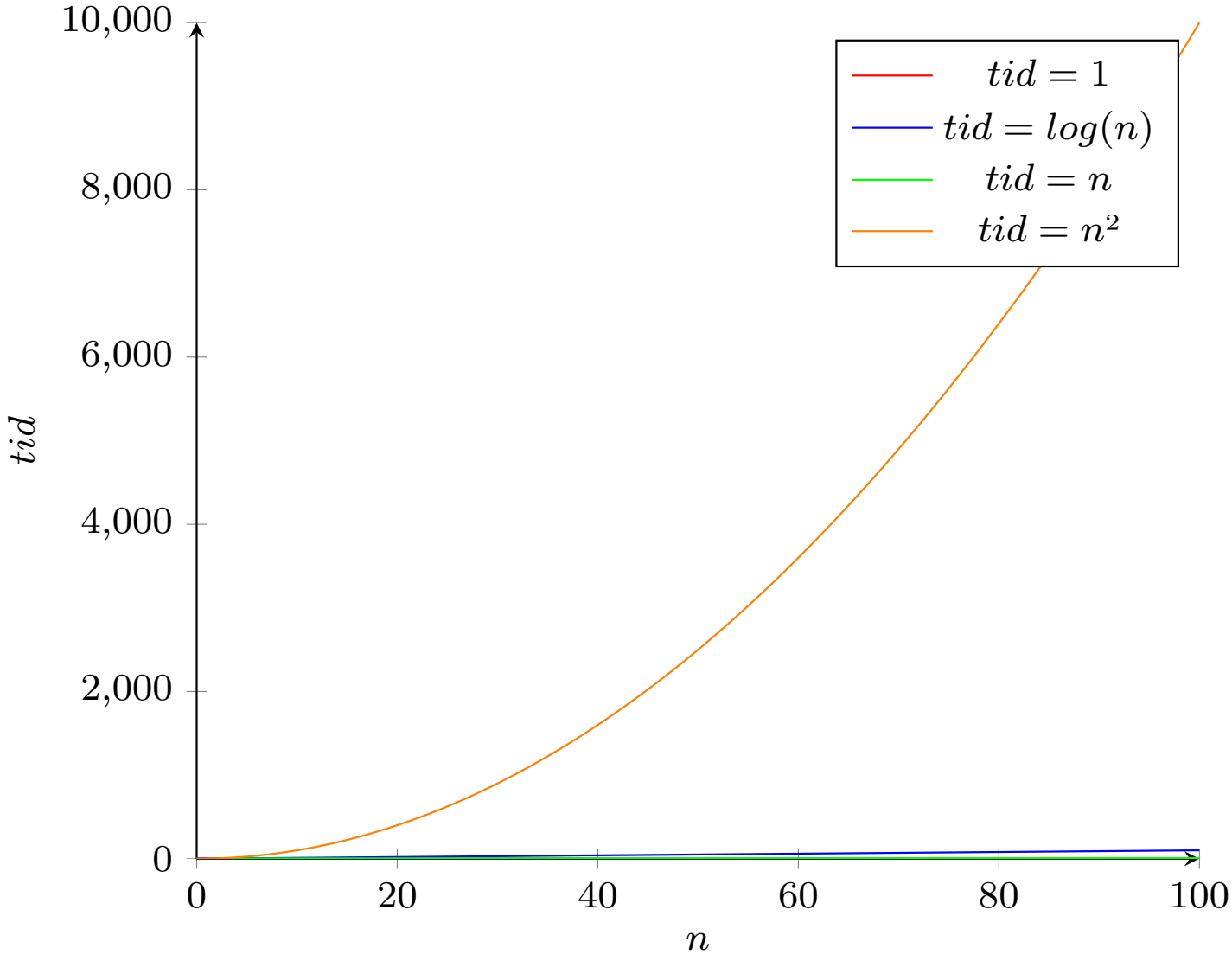
Intuition:

Slutsatser

Intuition:
Dubblering av steget mot målet istället för att ta ett konstant antal steg
$\implies \mathcal{O}(\log(n))$.
Om vi halverar mängden arbete i varje steg (matematiskt, sökpartition ...)
$\implies \mathcal{O}(\log(n))$.

Körtiden kommer att växa logaritmiskt med storleken på indata.
Alltså:
Om det tar 7 tidsenheter att exekvera funktionen då n $= 1024$, bör det ta ungefär 8 tidsenheter att exekvera densamma då n $= 2048$.
Återigen bör det inte spela någon roll om vi måste ta "några steg" för att komma fram till halveringen, eller dubbleringen (tänk lektionsuppgiften där vi kommer in halveringskedjan så fort vi når en tvåpotens).

Jämförelse av funktionerna

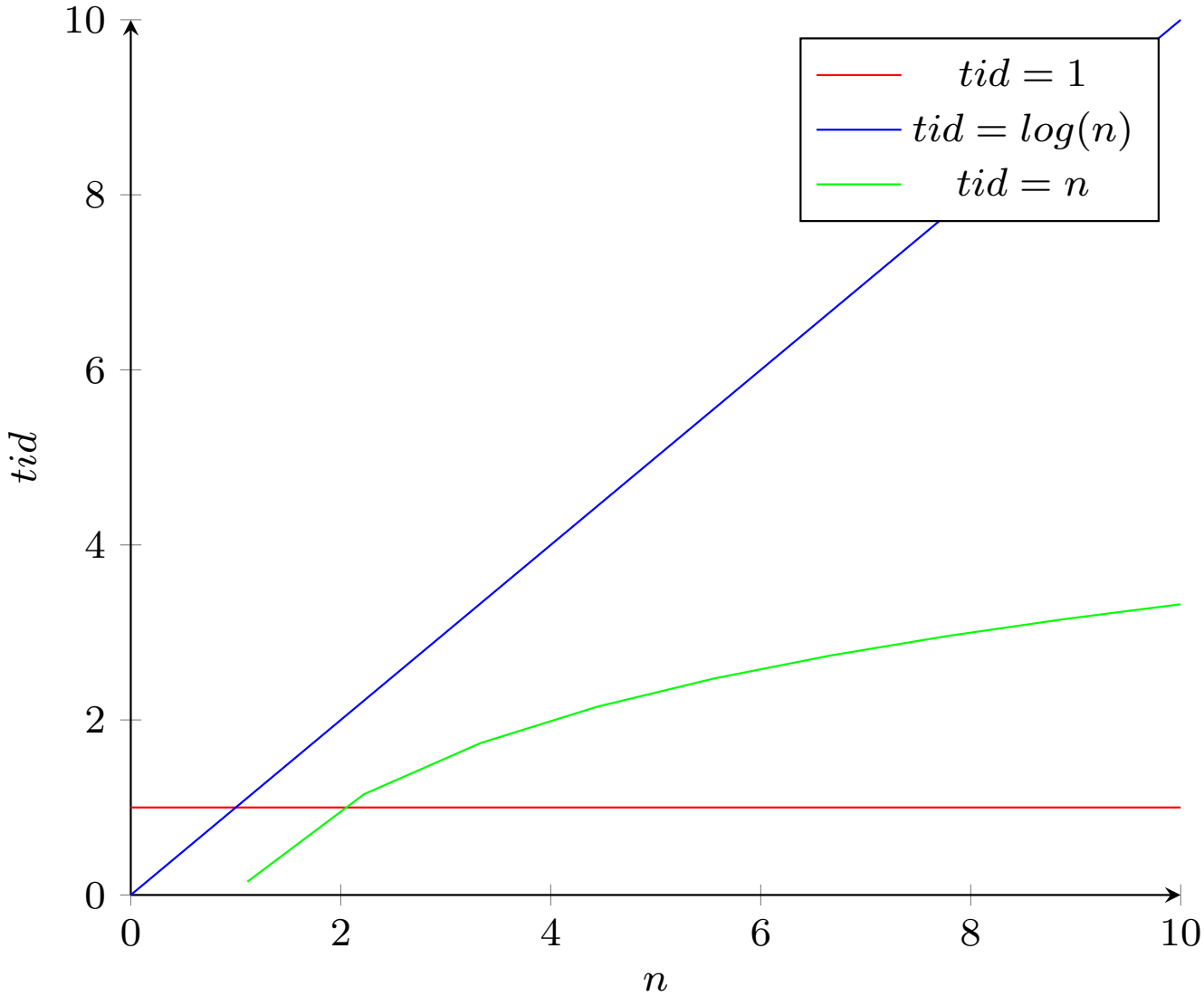


Slutsatser

Redan vid väldigt små n (n $= 100$) är n^2 så mycket större än de andra att vi knappt kan se de andra graferna.

Alltså: Oavsett konstanter är n^2 en mycket snabbare växande funktion än de andra.

Jämförelse av funktionerna



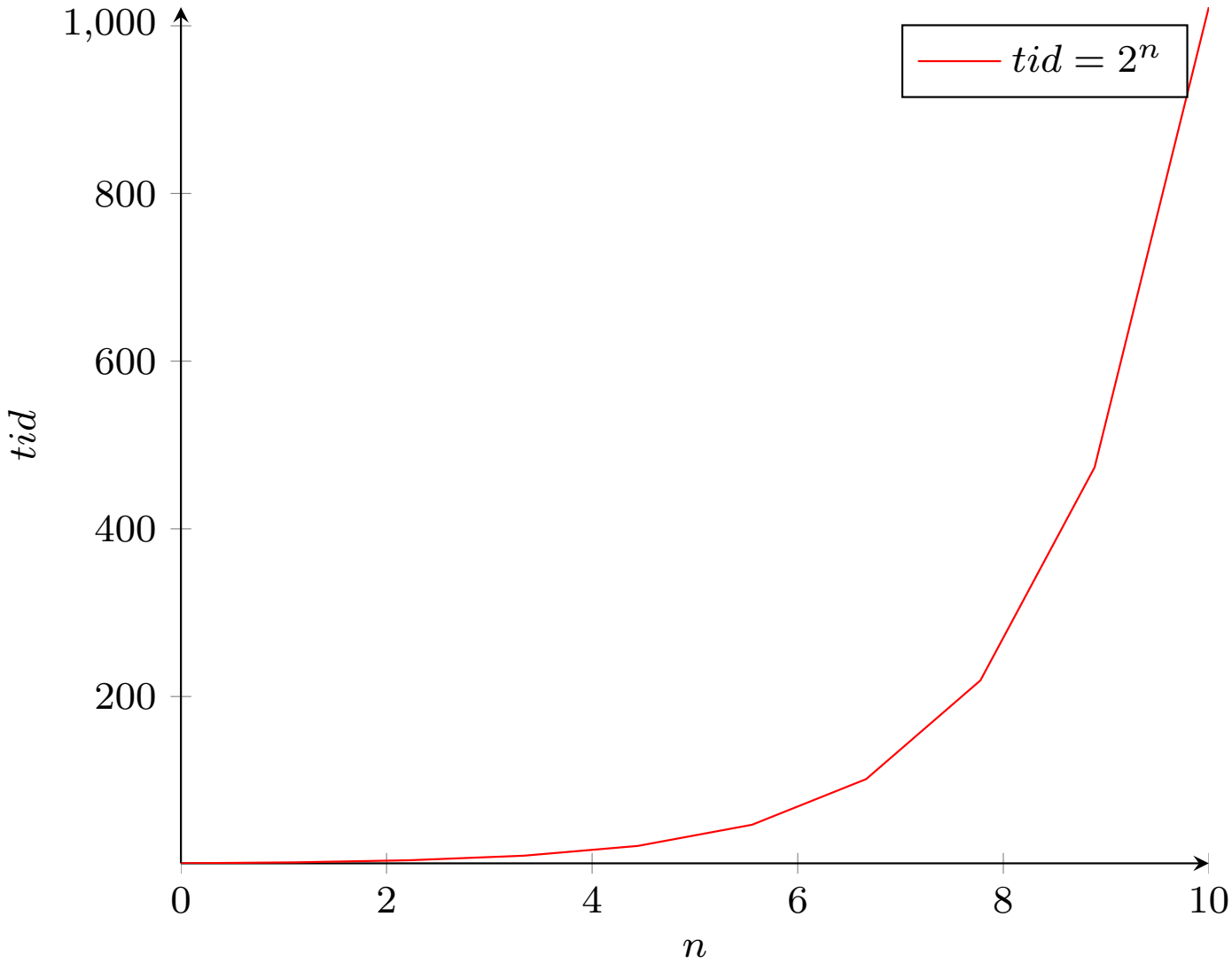
Slutsatser

Redan vid väldigt små n (n $= 10$) kan vi se att den linjära funktionen snabbt växer förbi den logaritmiska funktionen. Den logaritmiska växer mycket långsamt, men den växer till skillnad från den konstanta.
Alltså: Oavsett konstanter är $\mathcal{O}(n)$ en mycket snabbare växande funktion än de andra två vid riktigt stora n.

Eftersom den logaritmiska funktionen växer alls, till skillnad från den konstanta, är den "större" än den konstanta vid mycket stora n.

LINKÖPING UNIVERSITY

$O(2^n)$

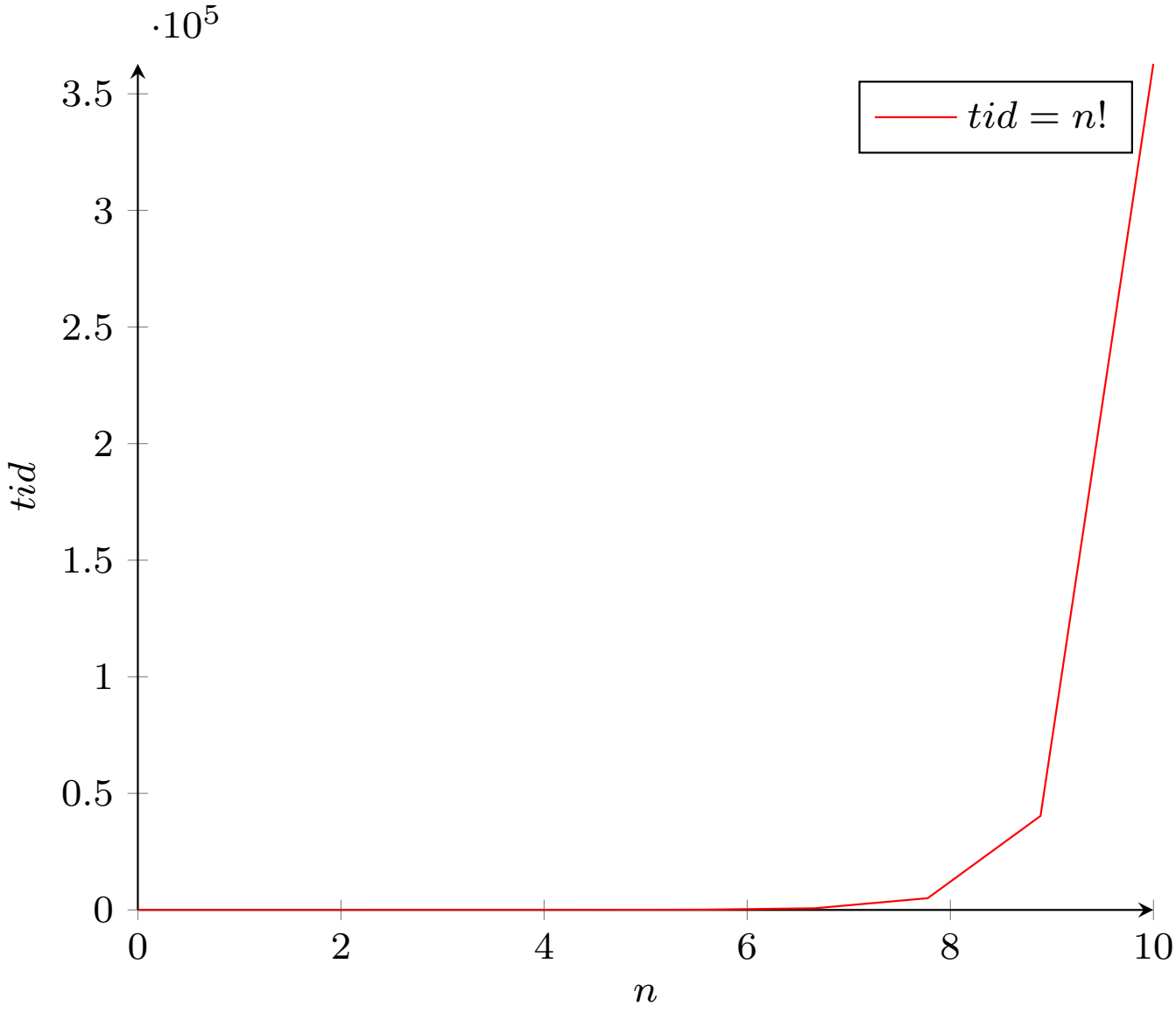


Exempel

Förekommer i synnerhet i rekursiva algoritmer som bryter ned ett problem av storlek n till två problem av storlek $\approx n - 1$.
Ex: Naiv beräkning av n'te fibonacci-talet:

```
int fib(int n) {
  if (n == 1 || n == 0) {
    return 1;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

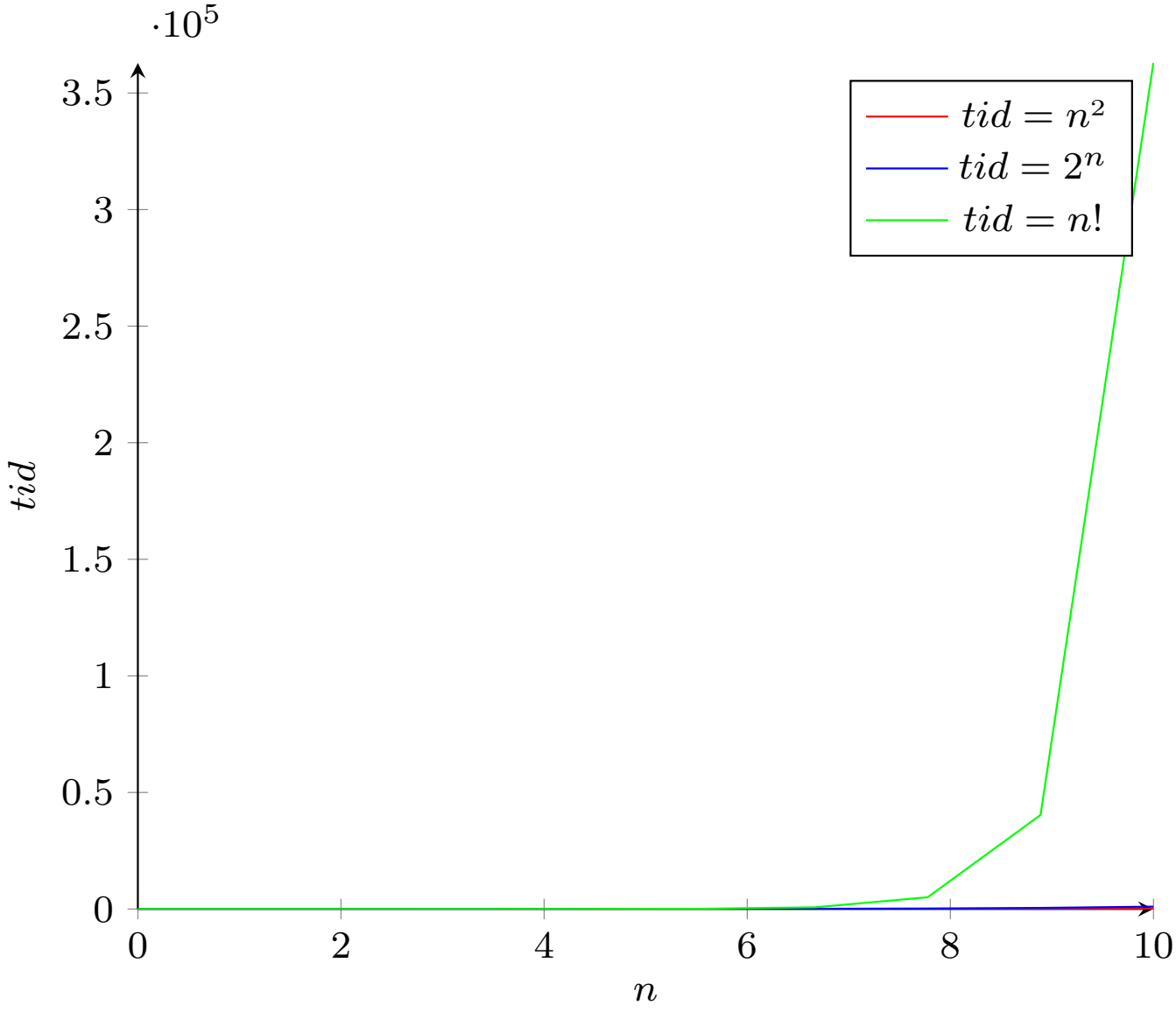
$O(n!)$



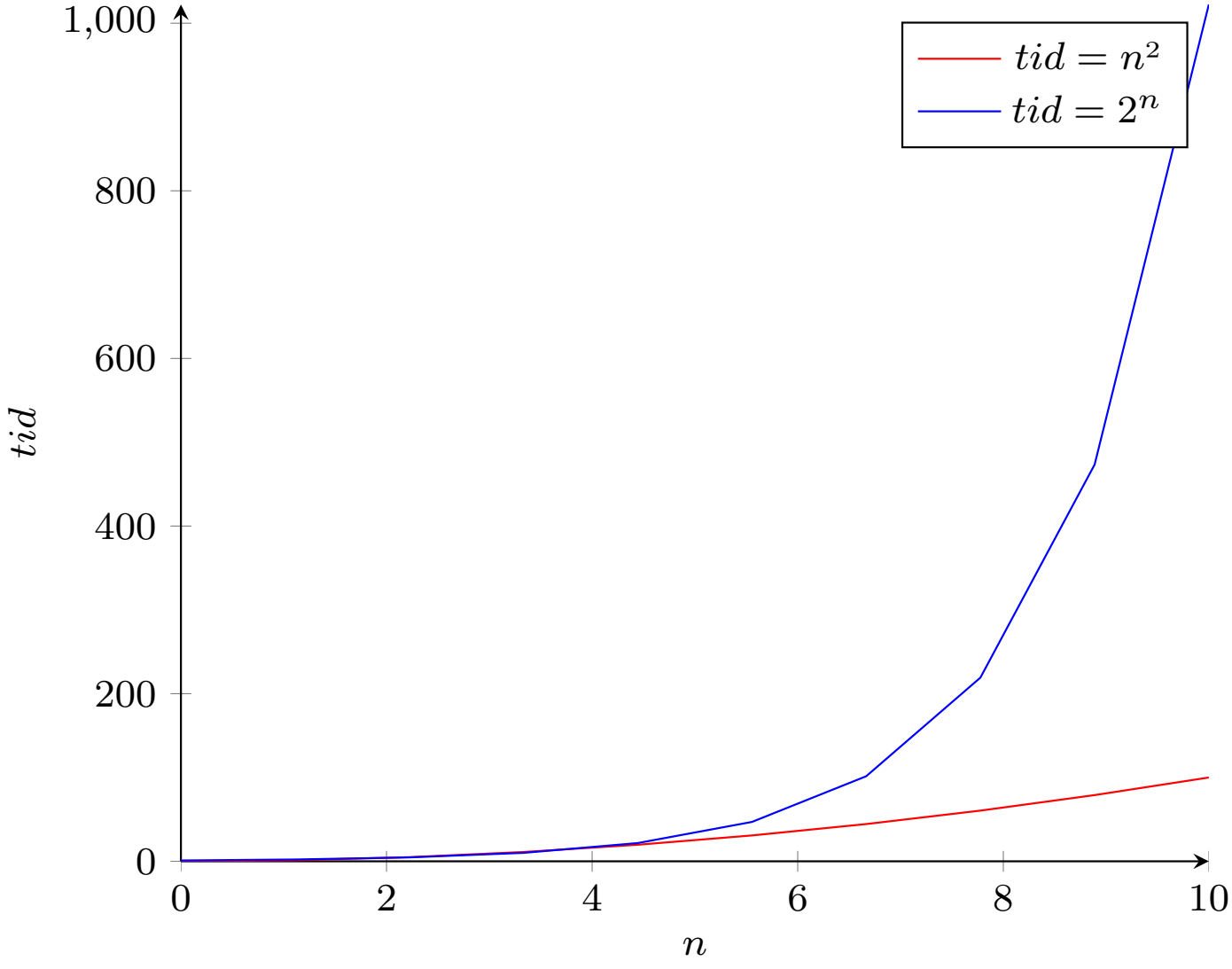
Exempel

Hitta alla permutationer av en sträng, array, ...
Generellt en kombination av loopar och rekursiva anrop, något ni inte kommer att behöva analysera i denna kurs.
Vi kan dock se att $n!$ sticker iväg otroligt snabbt jämfört med 2^n.

Stora funktioner



Stora funktioner



Slutsatser

$n!$ växer enormt mycket snabbare än de andra två stora funktionerna, och vi ser redan på små n att 2^n snabbt växer ifrån n^2.

Sammanfattning:
$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^x), x > 2 \subset \mathcal{O}(2^n) \subset \mathcal{O}(n!)$.

Nästa gång:
Vi ses på föreläsning...

www.liu.se

LINKÖPING UNIVERSITY